Verification of Sequential Imperative Programs in Isabelle/HOL

Norbert Schirmer

Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

Verification of Sequential Imperative Programs in Isabelle/HOL

Norbert Schirmer

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Helmut Seidl
Prüfer der Dissertation:	1. Univ.-Prof. Tobias Nipkow, Ph.D.
	2. Univ.-Prof. Dr. Wolfgang J. Paul Universität des Saarlandes

Die Dissertation wurde am 31.10.2005 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 21.04.2006 angenommen.

Kurzfassung

Ziel der Dissertation ist es, eine Verifikationsumgebung für sequentielle imperative Programme zu schaffen. Zunächst wird unabhängig von einer konkreten Programmiersprache ein allgemeines Sprachmodell entwickelt, das ausdrucksstark genug ist um alle gängigen Programmiersprachkonzepte abzudecken: Gegenseitig rekursive Prozeduren, abrupte Terminierung und Ausnahmebehandlung, Laufzeitfehler, lokale und globale Variablen, Zeiger und Halde, Ausdrücke mit Seiteneffekten, Zeiger auf Prozeduren, partielle Applikation, dynamischer Methoden Aufruf und unbeschränkter Indeterminismus.

Für dieses Sprachmodell wird eine Hoare Logik sowohl für partielle alsauch für totale Korrektheit entwickelt. Darauf aufbauend wird ein Verifikations-Bedingungs-Generator implementiert. Die Hoare Logik erlaubt die Integration von statischer Programmanalyse und Software Model Checkern in die Verifikation.

Desweiteren wird eine Teilsprache von C in die Verifikationsumgebung eingebettet, um die Durchgängigkeit zu einer realen Programmiersprache zu demonstrieren.

Die gesamte Entwicklung wurde im Theorembeweiser Isabelle durchgeführt. Dadurch wird zum einen die Korrektheit maschinell sichergestellt und zum anderen steht nun für die Verifikation von Programmen die reichhaltige Infrastruktur einer vollwertigen und universellen Beweisumgebung zur Verfügung.

Abstract

The purpose of this thesis is to create a verification environment for sequential imperative programs. First a general language model is proposed, which is independent of a concrete programming language but expressive enough to cover all common language features: mutually recursive procedures, abrupt termination and exceptions, runtime faults, local and global variables, pointers and heap, expressions with side effects, pointers to procedures, partial application and closures, dynamic method invocation and also unbounded nondeterminism.

For this language a Hoare logic for both partial and total correctness is developed and on top of it a verification condition generator is implemented. The Hoare logic is designed to allow the integration of program analysis or software model checking into the verification.

To demonstrate the continuity to a real programming language a subset of C is embedded into the verification environment.

The whole work is developed in the theorem prover Isabelle. Therefore the correctness is machine-checked and in addition the rich infrastructure of the general purpose theorem prover Isabelle can be employed for the verification of imperative programs.

Funk is, what you don't play.
— Maceo Parker

Acknowledgements

I thank Tobias Nipkow for supervising this thesis and giving me the opportunity to work in his research group. I thank Wolfgang J. Paul for acting as referee and initiating the Verisoft project.

I am indebted to Amine Chaieb, Gerwin Klein, Martin Strecker and Martin Wildmoser for reading and commenting on drafts of this thesis.

I also thank all the former and current members of the Isabelle team in Munich for the friendly and relaxed working atmosphere, where there is always time for a coffee break and discussions: Clemens Ballarin (for his wise advise), Gertrud Bauer, Stefan Berghofer (for all his support with Isabelle), Amine Chaieb, Florian Haftmann, Gerwin Klein (for collaboration and friendship even across the continents), Alexander Krauss, Farhad Mehta, Steven Obua (for his power), David von Oheimb, Leonor Prensa-Nieto, Sebastian Skalberg, Martin Strecker (for introducing me to theorem proving and answering all my questions), Tjark Weber, Markus Wenzel (for his ideal quest for *proper* solutions), and Martin Wildmoser.

I thank my parents Anneliese and Jörg, and my girlfriend Sarah Erath for their love and support.

Contents

CHAPTER 1

Introduction

Contents

1.1 Motivation

Software correctness is an issue since the beginning of computer programming. Due to the omni-presence of computers and embedded devices in modern society, malicious software becomes a serious threat for economy and even for human lives. To improve software quality, today's software engineering provides methods to guide the process of software development. From the problem analysis and requirements phase, over the implementation phase until the deployment of the software. These methods mostly introduce informal or semi-formal ways to describe the requirements of the software to improve communication and to derive test cases. The main technical means to ensure software quality is still extensive program testing, which consumes a lot of the overall effort to build the software and only has a limited effect. As Dijkstra [29] pointed out: "Program testing can be used to show the presence of bugs, but never to show their absence!"

As alternative to testing Floyd [35], Hoare [47], Dijkstra [29] and others pioneered the idea of rigorous program verification. As opposed to testing, verification traces every possible program path as it works on a symbolic and abstract level. The method relies on mathematical proof to ensure program correctness. The approaches are labelled in the literature as *(Floyd-)Hoare logic*, or in case of Dijkstra's work as the *weakest precondition calculus* or *predicate transformers*. The basic idea is to describe a program state by a predicate, a so called *assertion*. The calculus defines rules, how a certain statement in the programming language affects the assertion. A piece of code is specified by two assertions, a pre- and a postcondition. If the initial state satisfies the precondition, then the final state, after execution the program, satisfies the postcondition. The rules of the calculus are syntax directed and allow to decompose the specification of a program to a mere logical proposition in the assertion logic, the so called *verification condition*. If the verification condition can be proven, then the program satisfies its specification. Since the rules of the calculus are syntax directed, the process of decomposing a specification to the verification condition can even be automated (provided that loops are annotated with an invariant). This is the purpose of a *verification condition generator*. This was quite a success, since it allows to reduce program verification to mere logical reasoning. However, it turned out that the verification conditions can get quite complicated. A lot of knowledge and theorems about the domain the programs work on is needed. This ranges from arithmetic for numeric applications, to structures like lists, trees, graphs or sets in case of pointer manipulating programs. To supply proper tool support for reasoning about the verification condition, a general purpose theorem prover is needed in order to deal with the comprehensive background theory. In recent years the technology of interactive theorem provers, based on an expressive logic, has reached a mature state with a wide range of successful applications, from pure logic [92, 82], over mathematics [38, 9] to programming language semantics [81, 58]. This technology is definitely well suited to reason about the verification condition. Furthermore, as the logic of the interactive theorem provers is expressive enough to formalise programming language semantics, it is even possible to derive the Hoare calculus within the theorem prover and hence ensure the soundness of the program logic. Gordon [71] was the first who followed this approach and embedded a simple while language into higher order logics.

To demand that the program calculus has to be derived in a theorem prover does not at all originate form a disproportionate need for security. As Apt pointed out in his survey on Hoare logics [7]: "various proofs given in the literature are awkward, incomplete, or even incorrect". A prominent example is the procedure call rule for Euclid. Euclid is a programming language that was developed in parallel with its Hoare logics [44, 64], since program verification was the driving force for the language design. Cartwright and Oppen [21] and later Gries [43] and Olderog [86] noticed that the proposed procedure call rule for Euclid is indeed unsound.

Besides these general soundness issues of a program logic, there are also some practical considerations within the context of the Verisoft[1] project that stimulated this work. The Verisoft project aims at the *pervasive* verification of complete computer systems, comprising the hardware, the operating system and distributed user level applications. The main programming language for the software layers is C0, a type-safe subset of C. The user level applications, as well as large parts of the operating system can be written in C0. However, parts of the operating system have to be written with in-line assembler code. Those parts modify the state of the computer in areas that are usually invisible for a C0 program. For instance the operating system can increment or decrement the allocated memory of another user program. Hence not all parts of the operating system can be verified on the C0 layer. However, since large parts of the operating system are indeed written in C0 it is not desirable to conduct the whole verification on the low level of the assembler language. Fortunately this is not necessary as long as the complete meta theory for the program logic is available. The soundness theorem of the Hoare logic can be employed to transfer proven program properties to the C0 semantics. Furthermore, a compiler correctness theorem between the C0 semantics and the assembler

[1]The Verisoft project is funded by the German Federal Ministry of Education and Research (BMBF) under grant 01 IS C38. Verisoft home page: `http://www.verisoft.de`.

semantics makes the program properties available on the assembler layer. Hence the formal soundness theorem for the Hoare logic is not only a desirable add-on, but indispensable to actually transfer program properties to the lower layers for further reasoning.

1.2 Contributions

The focus of this thesis is to provide a sound and practically useful verification environment for sequential imperative programs. First I develop a general language model for sequential imperative programs called Simpl. It is not restricted to a particular programming language, but covers all common language features of imperative programs: mutually recursive procedures, global and local variables, pointers and heap, expressions with side effects, runtime faults like array bound violations or dereferencing null pointers, abrupt termination like `break`, `continue`, `return` in C and even exceptions, pointers to procedures, partial application and closures, dynamic method invocation and also unbounded nondeterminism. For this language I define an operational semantics and a Hoare logic for both partial and total correctness. These Hoare logics are proven sound and complete with respect to the operational semantics. All the formalisation and proofs are conducted in the theorem prover Isabelle/HOL. This is the first machine checked completeness proof for total correctness of a Hoare logic for such an expressive language. Moreover, the handling of auxiliary variables and the consequence rule are clarified. It turns out that there is no need to mention auxiliary variables at all in the core calculus.

The application of the Hoare logic rules is automated in a verification condition generator, implemented as Isabelle tactic. This rules out any soundness concerns of the implementation of the verification condition generator. Moreover, the handling of procedure definitions and specifications is seamlessly integrated into Isabelle. This makes the comprehensive infrastructure of Isabelle available for program verification and leads to a flexible, sound and practically useful verification environment for imperative programs.

The Hoare logic is extended to facilitate the integration of automatic program analysis or software model checking into the verification environment. The properties that the program analysis infers can be added as assertions into the program. These assertions are treated as granted for the rest of the verification. The Hoare calculus and the notion of validity is adapted to make sound reasoning about those assertions possible. The analysis result is again captured in a Hoare triple. This leads to a clear and declarative interface of the Hoare logic to program analysis.

To demonstrate the connection to a real programming language, a type-safe subset of the C programming language, called C0 is formally embedded into Simpl. This embedding illustrates how the type safety result for C0 can be exploited to switch to a simpler state model for the verification of individual (welltyped) C0 programs. For example, primitive types of C0 are directly mapped to HOL types in Simpl. Hence type inference of Isabelle takes care of the basic typing issues. Moreover, the heap model of Simpl already rules out aliasing between pointers of unequal type or to different structure fields. This is the first time this model is formally justified. Moreover, the embedding of C0 in Simpl shows how a deep embedding, tailored for meta theory, can be transferred to a shallow embedding for the purpose of program verification. The correctness proof of the embedding allows to transfer the program properties that are proven on the Simpl level back to the original C0 semantics.

1.3 Related Work

The related work can be grouped in three categories.

- General work about Hoare logics,
- formalisation of programming languages and program logics in theorem provers, and
- tools for program verification in general.

The decisive characteristic of the work in this thesis is that all the meta theory is developed and proven in Isabelle/HOL [80] and the actual program verification also takes place in Isabelle/HOL. So everything is machine checked and there are no gaps that can introduce any soundness issues. Moreover, Simpl makes no restrictions on the state space model and on the basic actions of the programming language. These can be exchanged and adapted to fit to a concrete programming language and to the current verification task.

As Hoare logic was already invented in 1969 there is a vast amount of literature available on this topic. Already in 1981 Apt wrote a first survey article [7]. Lots of the work made its way into textbooks [76, 118, 8, 36] and can even be regarded as folklore in computer science. Therefore I only mention the work that is closely related to this thesis. In this work I present the soundness and completeness proofs for a Hoare logic for partial and for total correctness. Simpl features mutually recursive procedures as well as unbounded nondeterminism. Let us first sketch some history of Hoare logics for deterministic languages with procedures. Hoare proposed a calculus [48] that was proven sound and complete by Olderog [86]. In his survey article [7] Apt presents a sound and complete Hoare logics for both partial and total correctness. For partial correctness he follows Gorelick [41] and for total correctness he completes the work of Sokolowski [106]. Later America and de Boer [3] find that the system for total correctness was unsound. They modify the system and present new soundness and completeness proofs. However, their calculus suffers from three additional adaptation rules. Kleymann (formally named Schreiber) [104] subsumes all three rules by using a consequence rule that goes back to Morris [73]. Kleymann has also formalised his work in the theorem prover LEGO [97]. For the first time the soundness and completeness of a Hoare logic have been proven in a theorem prover. Oheimb [84] builds on the work of Kleymann and presents a sound and complete Hoare logic for partial correctness for a significant subset of Java. Nipkow [77] simplifies some aspects of Kleymann's proof system and extends the proofs for partial and total correctness to procedures in the context of unbounded nondeterminism. Nipkow relates his results to the work of Apt [5] and Apt and Plotkin [6] on unbounded nondeterminism. He identifies the main differences in that they use ordinals instead of well-founded relations and do not consider procedures. Both Oheimb and Nipkow have formalised their work in Isabelle/HOL. The work in this thesis introduces some further simplification to the proof system of Kleymann and extends the work of Nipkow to Simpl. In particular Simpl supports abrupt termination and can deal with procedures with parameters, dynamic method invocation and higher order features like pointers to procedures and closures.

The tradition of embedding a programming language in HOL goes back to the work of Gordon [71], where a while language with variables ranging over natural numbers is introduced. A polymorphic state space was already used by Wright *et al.* [111] in their mechanisation of refinement concepts, by Harrison in his formalisation of Dijkstra [46], and by Prensa to verify parallel programs [98]. Still procedures were not present. Homeier [50] introduces procedures, but the variables are again limited to numbers. Later on detailed semantics for Java [84, 103, 102, 51] and C [81] were embedded in a theorem prover. Unfortunately verification of even simple programs suffers from the complex models. The problem is that the models are biased towards meta theory of the programming language. Although properties like type safety of the programming language can be proven once and for all, typing issues are explicit in the model and permanently show up in the verification of individual welltyped programs. The meta theory of Simpl completely abstracts from the state space representation. The state space can be instantiated for every individual program. With this approach the types of programming language variables can be mapped to HOL types, and moreover a shallow embedding of expressions can be used. All this simplifies the verification of individual programs.

The Why tool [33] implements a program logics for annotated functional programs (with references) and produces verification conditions for external theorem provers. It can handle uninterpreted parts of annotations that are only meaningful to the external theorem prover. With this approach it is possible to map imperative languages like C to the tool by representing the heap in reference variables. Although the Why tool and the work we present in this paper both provide comparable verification environments for imperative programs the theoretical foundations to achieve this are quite different: Filliâtre builds up a sophisticated type theory incorporating an effect analysis on the input language, whereas the framework of Hoare logics and the simple type system of HOL is sufficient for our needs. Moreover, the entire development in this thesis, the calculus together with its soundness and completeness proof, is carried out in Isabelle/HOL, in contrast to the pen and paper proofs of Filliâtre [32]. The formal model in Isabelle/HOL allows to reason about the embedding of a programming language to Simpl. In contrast, the embedding of C [34] and Java [65] to the Why tool have to be trusted.

The following tools for the verification of imperative programs all have in common that they are less foundational than the approach presented in this thesis. The tools and their meta theory are not developed in a uniform logical framework like HOL: The Jive tool [70] implements a Hoare logic for Java [96]. The resulting verification conditions are passed to an interactive theorem prover, currently Isabelle/HOL. The KIV-tool [100] uses dynamic logic [45] for program verification. Recently Stenzel has extended it to handle Java Card programs [108, 107]. Stenzel has also verified this extension within the KIV-tool. The KeY-tool [2] is also based on dynamic logic to reason about Java. The B-Method [1] and VDM [55] focus on building programs in a step-wise refinement process.

Several works propose the integration of automatic tools into interactive theorem proving. In the context of hardware verification Pisini *et al.* [31], Rajan *et al.* [99] and Amjad [4] have integrated model checkers. Similarly Joyce and Seger [56] proposed a link between symbolic trajectory evaluation and interactive theorem proving. On the other hand there is work on the integration of general purpose automatic theorem provers for fragments of HOL, like first order theorem provers [69] or SAT solvers [113] and arithmetic decision procedures [11]. However, I am not aware of any work to integrate program analysis or *software* model checking in a Hoare logic based framework. On the level of byte-code, Wildmoser *et al.* [117] propose a similar approach. It allows to integrate static analysis results for Java byte-code into a proof carrying code framework.

The approach followed in context of the Why tool is complementary to the one in this thesis. The Why tool can generate verification conditions for several theorem provers. These include the automatic theorem prover Simplify[2] [75] and the interactive theorem prover Coq [14]. In some cases the verification condition can already be proven by Simplify, otherwise it has to be proven in Coq. In our setting a similar effect could be achieved by integrating Simplify as an oracle in Isabelle that is invoked on the verification condition. In contrast, the approach in this thesis integrates the results of the program analysis or software model checker *before* the verification condition generator is invoked. We do not expect the automatic tool to solve the complete verification condition, but can exploit the assertions it returns already during verification condition generation and also in the following interactive verification.

1.4 Overview

For the rest of this chapter I introduce preliminaries on Isabelle and the notational conventions of this thesis.

Chapter 2 introduces Simpl, a model for sequential imperative programming languages. The formal syntax and semantics of Simpl is defined and a couple of examples illustrate how common language features can be expressed in Simpl.

In Chapter 3 a Hoare logic for partial and total correctness of Simpl programs is developed and proven sound and complete.

In Chapter 4 a verification condition generator is built on top of the Hoare logic and a series of examples illustrate how various aspects of the verification of imperative programs are handled.

Chapter 5 discusses how program analysis or software model checking can be integrated into the verification environment.

Chapter 6 studies the compositionality of the calculus and how verified libraries can be built and reused.

Chapter 7 introduces C0, a type-safe subset of the programming language C. Its syntax and semantics is defined and a type system and a definite assignment analysis is developed. A couple of type soundness theorems are proven.

In Chapter 8 C0 is embedded into Simpl and this embedding is proven correct. The final correctness theorem allows to transfer program properties from Simpl back to the original C0 program. An example concludes the presentation.

Chapter 9 reports on the practical experiences with the verification environment and finally concludes the thesis with a summary and pointers to further work.

Appendix A presents a collection of theorems that are omitted in Chapter 2.

[2]See the Simplify home page: `http://research.compaq.com/SRC/esc/Simplify.html`

1.5 Preliminaries on Isabelle

Isabelle [90] is a generic proof assistant. It provides a framework to declare deductive systems, rather than to implement them from scratch. Currently the best developed object logic is HOL [80], higher order logic, including an extensive library of (concrete) mathematics, as well as various packages for advanced definitional concepts like (co-)inductive sets and types, primitive and well-founded recursion etc. To define an object logic, the user has to declare the syntax and the inference rules of the object logic. By employing the built-in mechanisms of Isabelle/Pure, higher-order unification and resolution in particular, one already gets a decent deductive system. For sizable applications some degree of automated reasoning is essential. Existing tools like the classical tableau prover or conditional rewriting can by instantiated by a minimal ML-based setup. ML [91] is the implementation language of Isabelle. Moreover, Isabelle follows the well-known *LCF system approach* [40], which allows us to write arbitrary proof procedures in ML without breaking system soundness since all those procedures are expanded into primitive inferences of the logical kernel.

Isabelle's meta logic [89], which is minimal higher-order logic with connectives $\bigwedge$ (universal quantification), $\Longrightarrow$ (implication), and $\equiv$ (equality), is used to describe natural deduction style inference rules and basic definitions. The Isabelle kernel manipulates formulas on the level of the meta logic.

For example, the introduction rule for conjunction:

$$\frac{P \qquad Q}{P \wedge Q}\ (\textsc{conjI}),$$

is expressed as:

$$[\![P; Q]\!] \Longrightarrow P \wedge Q,$$

in the meta logic. The brackets $[\![\ldots]\!] \Longrightarrow$ separate the premises from the conclusion. They are syntactic sugar for nested entailment. Without these brackets the rule reads as follows:

$$P \Longrightarrow (Q \Longrightarrow P \wedge Q).$$

Isabelle supports two kinds of proof styles. A tactic style and a declarative, human-readable style. In the tactic style the current proof goal is manipulated with so called tactics. These tactics range from the application of a single inference rule to automatic methods, like decision procedures for linear arithmetic, rewriting or a classical tableau prover. The effect of a tactic is an altered goal state. It can either solve the goal completely, split the goal to various subgoals, or just modify or simplify it. Here is an example proof.

lemma $P \wedge Q \Longrightarrow Q \wedge P$

1. $P \wedge Q \Longrightarrow Q \wedge P$

The command **lemma** initiates the initial subgoal.

apply (*rule conjI*)

1. $P \wedge Q \Longrightarrow Q$
2. $P \wedge Q \Longrightarrow P$

The rule *conjI* is applied backwards. This means its conclusion is unified with the current subgoal and the premises result in the new subgoals.

apply (*erule conjE*)

1. $[\![P; Q]\!] \Longrightarrow Q$
2. $P \wedge Q \Longrightarrow P$

The elimination rule *conjE* splits the conjunction in the premises. The first subgoal can now be solved by assumption.

apply *assumption*

1. $P \wedge Q \Longrightarrow P$

And analogous for the remaining subgoal.

apply (*erule conjE*)

1. $[\![P; Q]\!] \Longrightarrow P$

apply (*assumption*)
done

The major drawback of tactic style proofs is that a reader can only understand them if he can either imagine or see the current goal state. This also complicates maintenance of the theories.

The alternative, declarative proof style, also named (proper) Isar style [114, 116, 79], is more verbose and explicit about the objects that are manipulated. The proof is self contained. You do no longer need a goal state to follow the argumentation. Without going into details, here is the same lemma in the Isar style.

lemma $P \wedge Q \Longrightarrow Q \wedge P$
proof –
  assume $P \wedge Q$
  then obtain Q **and** P **..**
  then show $Q \wedge P$ **..**
qed

If a lot of theorems depend on the same set of assumptions, this context can be grouped together in a so called **locale** [10]. An algebraic example is reasoning about groups. An abstract group fixes the operations for product and inverse and an identity element, together with the axioms of associativity, left-inverse and the left-identity.

The following definition of locale *group-context* encapsulates the local parameters (with local syntax) and assumptions. The type *'a* is a type variable for elements of the group. The infix type constructor $\Rightarrow$ is for the total function space. So the parameter *prod* is a function that takes two arguments. In the locale the syntax $x \cdot y$ can be used.

locale *group-context* =
  fixes *prod* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** · *70*)
    and *inv* :: $'a \Rightarrow 'a$ $((-^{-1})$ [*1000*] *999*)
    and *one* :: $'a$ (1)
  assumes *assoc*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
    and *left-inv*: $x^{-1} \cdot x = 1$
    and *left-one*: $1 \cdot x = x$

We may now prove theorems within a local context just by including a directive "(**in** *name*)" in the goal specification. The final result is stored within the named locale, still holding the context. For instance, here is the proof that the right-inverse is derivable form the group axioms. The … abbreviate the last mentioned term. In this proof it is always the right hand side of the equation above. The **also** triggers transitivity reasoning for the equations.

theorem (**in** *group-context*)
  right-inv: $x \cdot x^{-1} = 1$
proof –
  have $x \cdot x^{-1} = 1 \cdot (x \cdot x^{-1})$ **by** (*simp only*: *left-one*)
  also have … $= 1 \cdot x \cdot x^{-1}$ **by** (*simp only*: *assoc*)
  also have … $= (x^{-1})^{-1} \cdot x^{-1} \cdot x \cdot x^{-1}$ **by** (*simp only*: *left-inv*)
  also have … $= (x^{-1})^{-1} \cdot (x^{-1} \cdot x) \cdot x^{-1}$ **by** (*simp only*: *assoc*)
  also have … $= (x^{-1})^{-1} \cdot 1 \cdot x^{-1}$ **by** (*simp only*: *left-inv*)
  also have … $= (x^{-1})^{-1} \cdot (1 \cdot x^{-1})$ **by** (*simp only*: *assoc*)
  also have … $= (x^{-1})^{-1} \cdot x^{-1}$ **by** (*simp only*: *left-one*)
  also have … $= 1$ **by** (*simp only*: *left-inv*)
  finally show *?thesis* **.**
qed

The theorems in locale *group-context* build the abstract theory of groups. Isabelle also allows to instantiate a locale as we want to access the theorems for a concrete group, for example, the real numbers with addition. The abstract operations are instantiated with the corresponding operations for reals and the group axioms have to be proven for these operations. Then Isabelle automatically instantiates all the theorems that are accumulated in locale *group-context* for the reals. Moreover, locales can extend other ones. Like an abelian group adds the commutativity axiom to groups:

locale *abelian-group-context* = *group-context* +
  assumes *commute*: $x \cdot y = y \cdot x$

Of course, all the theorems for groups are automatically available for abelian groups. Locales support a modular development of the theories. The abstract theory can be developed independent of concrete instances. And every concrete instance can import the abstract theory.

Currently the best developed object logic of Isabelle is HOL, an encoding of higher order logic, augmented with facilities for defining data types, records, inductive sets as well as primitive and total general recursive functions. The work in this thesis is based on Isabelle/HOL. Quoting from the Isabelle/HOL tutorial [80]:

HOL = Functional Programming + Logic.

Hence the notation in this thesis is a mixture of functional programming and standard mathematical-logical conventions. The main impact from functional programming is that functions are usually defined in a curried form and function definitions for data-types use pattern matching. The motivation for the curried form is that partial application can be exploited. For example, instead of defining an addition function *add* of type $int \times int \Rightarrow int$, it is defined as $int \Rightarrow int \Rightarrow int$, which means $int \Rightarrow (int \Rightarrow int)$. Hence an increment function can be defined as *add 1*, since this partial application results in a function of type $int \Rightarrow int$. The curried style of functions together with partial application is also the reason why the arguments of a function are not grouped together with parenthesis. An application *add* (i, j) corresponds to a signature $int \times int \Rightarrow int$, as (i, j) is a pair in HOL. The parameters of a function are just separated by a white space: *add i j*. Function application binds tighter than any infix or mixfix operation. For instance, $f\ i + g\ i\ j$ means $(f\ i) + (g\ i\ j)$.

After this short introduction about the principles of the notation, we systematically introduce some more notation and basic (data) types and their primitive operations.

Types The basic types are truth values (*bool*), natural numbers (*nat*) and integers (*int*). The space of total functions is denoted by the infix $\Rightarrow$. Type variables are written as *'a*, *'b*, *'c*, etc. The notation $t :: \tau$ means that HOL term t has HOL type τ.

The functions $nat :: int \Rightarrow nat$ and $int :: nat \Rightarrow int$ convert between natural numbers and integers. Negative integers are mapped to natural number *0*.

Logical Connectives The logical connectives are as usual: $\wedge$, $\vee$, $\longrightarrow$, $\neg$, $\forall$ and $\exists$. Where the bound variables of quantifiers are separated from the body by a ".". For instance, $\forall x\ y.\ x < y$. Note that the two implications $\longrightarrow$ and $\Longrightarrow$ and the universal quantifies $\forall$ and $\bigwedge$, taken from HOL and the meta logic are equivalent. Since an HOL formula is an atomic entity for the meta logic, the scope of HOL quantifiers never extends over meta logical connectives. For example, in $\forall x.\ P\ x \Longrightarrow P\ x$ the x in the conclusion is not in scope of the universal quantifier. Of course the universal quantifier of the meta logic extends over meta logical connectives. For instance, in $\bigwedge x.\ P\ x \Longrightarrow P\ x$ both x are bound by the quantifier.

HOL defines a polymorphic equality =. Hence the "if and only if" is just Boolean equality in HOL.

As usual, the connectives $\longrightarrow$, $\Longrightarrow$, $\wedge$ and $\vee$ associate to the right, and $\wedge$ or $\vee$ bind tighter than $\longrightarrow$. For instance, the proposition $P \longrightarrow Q \longrightarrow R \longrightarrow S$ means $P \longrightarrow (Q \longrightarrow (R \longrightarrow S))$ and is hence equivalent to the proposition $P \wedge Q \wedge R \longrightarrow S$ (which is $(P \wedge Q \wedge R) \longrightarrow S$).

Furthermore, HOL provides a conditional, e.g. *if* $x < y$ *then* x *else* y , and for every data type there is a case distinction, e.g. *case* $x < y$ *of* $True \Rightarrow x \mid False \Rightarrow y$. Term abbreviations can be introduced as in *let* $x = t$ *in* u, which is equivalent to u where all occurrences of x have been replaced by t. For example, *let* $x = 1$ *in* $x + x$ is equivalent to $1 + 1$. Moreover, multiple bindings are separated by semicolons: *let* $x_1 = t_1;\ldots; x_n = t_n$ *in* u.

Pairs The type constructor for pairs is the infix $\times$. There are the two projections functions $fst :: 'a \times 'b \Rightarrow 'a$ and $snd :: 'a \times 'b \Rightarrow 'b$. Tuples are pairs nested to the right. So (a, b, c) is identical to $(a, (b, c))$ and also type $'a \times 'b \times 'c$ is identical to $'a \times ('b \times 'c)$.

Sets Sets of elements of type *'a* have type *'a set* . The notation for sets follows the usual mathematical convention, like $x \in A$, $A \cap B$, $A \cup B$, $-A$, $A \subseteq B$, etc. Only set comprehension is written as $\{x.\ P\ x\}$ instead of $\{x|\ P\ x\}$. The empty set is $\{\}$, a singleton set is, for instance, $\{a\}$, and the universal set is *UNIV*. A finite set expression like, for instance, $\{a, b, c\}$ abbreviates $\{a\} \cup \{b\} \cup \{c\} \cup \{\}$. Finite sets are characterised by the predicate *finite*, which is defined inductively by the following rules:

$$\frac{}{\mathit{finite}\ \{\}} \qquad \frac{\mathit{finite}\ A}{\mathit{finite}\ (\{a\} \cup A)}$$

Unions can be formed over the values of a given set. The syntax is $\bigcup_{x \in A} B$. The union over a type: $\bigcup_{x \in UNIV} B$, is abbreviated with $\bigcup_{x} B$. The union of a set of sets is expressed by $\bigcup C$.

The infix $`$ denotes the set image operation: $f\ `\ A = \{f\ a.\ a \in A\}$.

Lists Lists of elements of type *'a* have type *'a list*. The constructors of a list are the empty list [] and the infix constructor ·, which adds a single element to the front of the list. Moreover, $[a, b, c]$ abbreviates $a{\cdot}b{\cdot}c{\cdot}[]$. The infix @ appends two lists. The function *set* converts from lists to sets. Variable names ending in the plural "s" usually stand for lists, *length xs* yields the length of list *xs* and is abbreviated with $|xs|$, and $xs_{[n]}$, where $n{::}nat$, is the nth-element of *xs* (starting with 0). Moreover, with $xs[n := e]$ the list *xs* is updated at position *n* with value *e* . Term *distinct xs* means that the elements of *xs* are all distinct. The following standard functions from functional programming are also available in Isabelle/HOL.

With *rev xs* the list *xs* is reversed.

The function *map* applies a function to each element in a list:

$$map\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n].$$

The function *foldl* iterates a binary operation over a list:

$$foldl\ g\ e\ [x_1, x_2, \ldots, x_n] = (g\ (\ldots (g\ (g\ e\ x_1)\ x_2) \ldots)\ x_n).$$

For example, the sum of an integer list *is* can be calculated by *foldl* (+) *0 is*. The parenthesis around the infix operation + indicate that it is used as a normal (prefix) function here.

Function *zip* takes two lists and generates a list of pairs:

$$zip\ [x_1, x_2, \ldots, x_n]\ [y_1, y_2, \ldots, y_n] = [(x_1, y_2), (x_2, y_2), \ldots, (x_n, x_n)].$$

The input lists have to be of equal length. Otherwise the empty list is returned.

With *hd* the first element of a list is selected, and with *tl* the first element is removed from the list. Similarly function *last* selects the last element of a list, and *butlast* removes the last element from the list.

With *take n xs* the first *n* elements of the list *xs* are selected and with *drop n xs* the first *n* elements are dropped from *xs*.

With *replicate n e* a list with *n* copies of *e* is generated.

Function *concat* takes a list of lists and concatenates them:

$$concat\ [xs_1, xs_2, \ldots, xs_n] = xs_1\ @\ xs_2\ @\ \ldots\ @\ xs_n.$$

Functions The λ operator is used to define (anonymous) functions. For example, $\lambda x.\ x * x$ is a function that takes an argument x and squares it.

A function update is written $f(x := y)$ where $f :: 'a \Rightarrow 'b$, $x :: 'a$ and $y :: 'b$. It is defined as:

$$f(a := b) \equiv \lambda x.\ \textsf{if}\ x = a\ \textsf{then}\ b\ \textsf{else}\ f\ x.$$

Function composition is written as usual:

$$f \circ g \equiv \lambda x. f\ (g\ x).$$

Partial Functions HOL is a logic of total functions. To model partiality of functions there are two main approaches used throughout the HOL-Library and this thesis. First, the function can just return a default value within the range type. As types in HOL are non empty, every type has at least one element. Hence it is perfectly valid to declare a polymorphic constant *arbitrary*. This *arbitrary* is just a default value for each type. It is not defined but just declared. This means that it is an unspecified value of a type. A function can always return *arbitrary* in case of an undefined situation. For example, this is how *hd* is defined. Only for non-empty lists a reasonable result can be expected. The user only specifies the sole equation $hd\ (x{\cdot}xs) = x$. Internally the other case is defined as $hd\ [] = arbitrary$. The drawback of this approach is that one cannot distinguish between defined and undefined applications of *hd*. The result *arbitrary* can as well come from a legal application like $hd\ (x{\cdot}xs) = arbitrary$, since *arbitrary* could be equal to x. Hence most theorems about *hd* have the precondition $xs \neq []$. For example: $xs \neq [] \Longrightarrow hd\ xs{\cdot}tl\ xs = xs$.

In the second approach we explicitly adjoin a new element *None* to the type:

$$\textbf{datatype}\ 'a\ option = None \mid Some\ 'a$$

For succinctness we write $\lfloor a \rfloor$ instead of *Some a*. The under-specified inverse *the* of *Some* satisfies $the\ \lfloor x \rfloor = x$. Moreover, the function *option-map* applies a function to a defined value.

$$option\text{-}map \equiv \lambda f\ y.\ \textsf{case}\ y\ \textsf{of}\ None \Rightarrow None \mid \lfloor x \rfloor \Rightarrow \lfloor f\ x \rfloor$$

A partial function can be modelled as type $'b \Rightarrow 'a\ option$. In an undefined situation the element *None* is returned. The drawback is that the range type is no longer just $'a$ but $'a\ option$.

It depends on the situation and on sure instinct to decide which approach to choose.

Maps Maps are partial functions of type $'a \Rightarrow 'b\ option$, where *None* represents undefinedness and $f\ x = \lfloor y \rfloor$ means x is mapped to y. The domain of a map is defined as $dom\ m \equiv \{a.\ m\ a \neq None\}$. Instead of type $'a \Rightarrow 'b\ option$ we also write $'a \rightharpoonup 'b$. We abbreviate $f(x{:=}\lfloor y \rfloor)$ to $f(x \mapsto y)$. The latter notation extends to lists: $f([x_1,\ldots,x_m]\ [\mapsto]\ [y_1,\ldots,y_n])$ means $f(x_1{\mapsto}y_1)\ldots(x_i{\mapsto}y_i)$, where i is the minimum of m and n. This notation works for arbitrary list expressions on both sides of $[\mapsto]$, not just enumerations. Multiple updates like $f(x{\mapsto}y)(xs[\mapsto]ys)$ can be joined together as $f(x \mapsto y,\ xs\ [\mapsto]\ ys)$. The map $\lambda x.\ None$ is written *empty*, and $empty(\ldots)$, where $\ldots$ are updates, abbreviates to $[\ldots]$. For example, $empty(x{\mapsto}y,\ xs[\mapsto]ys)$ becomes the term $[x \mapsto y,\ xs\ [\mapsto]\ ys]$.

Overwriting map m_1 with map m_2 is written $m_1 \mathbin{++} m_2$ and is defined as:

$$m_1 \mathbin{++} m_2 \equiv \lambda k.\ \textsf{case}\ m_2\ k\ \textsf{of}\ None \Rightarrow m_1\ k \mid \lfloor v \rfloor \Rightarrow \lfloor v \rfloor.$$

Composition of map m_1 with map m_2 is written $m_1 \circ_m m_2$ and is defined as:

$$m_1 \circ_m m_2 \equiv \lambda k.\ \textsf{case}\ m_2\ k\ \textsf{of}\ None \Rightarrow None \mid \lfloor v \rfloor \Rightarrow m_1\ v.$$

Function *map-of* turns an association list, i.e. list of pairs, into a map:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathit{map\text{-}of} :: ('k \times 'v)\ list \Rightarrow ('k \rightharpoonup 'v)} \\ \mathit{map\text{-}of}\ [] & = & empty \\ \mathit{map\text{-}of}\ (p \cdot ps) & = & \mathit{map\text{-}of}\ ps(fst\ p \mapsto snd\ p) \end{array}$$

Note that with this definition the first elements may overwrite later occurrences of the same key:

$$\mathit{map\text{-}of}\ [(a, x), (a, y)]\ a = \lfloor x \rfloor.$$

The domain of a map m is restricted to a set A by the operation $m{\restriction}_A$:

$$m{\restriction}_A \equiv \lambda k.\ \textsf{if}\ k \in A\ \textsf{then}\ m\ k\ \textsf{else}\ None.$$

Two maps m_1 and m_2 satisfy the relation $m_1 \subseteq_m m_2$, if they agree on the domain of m_1:

$$m_1 \subseteq_m m_2 \equiv \forall k \in dom\ m_1.\ m_1\ k = m_2\ k.$$

Pattern Matching As in functional programming, (recursive) definitions are often defined with pattern matching on the data types. In those definitions the order of the equations is significant. Moreover, the dummy pattern - can be used. For example:

$$\begin{array}{lcl} f\ (x \cdot y \cdot xs) & = & \ldots \\ f\ (x \cdot xs) & = & \ldots \\ f\ \text{-} & = & \ldots \end{array}$$

The second equation is only applied to lists with only one element, since lists with at least two elements are already handled by the first equation. Similarly the third equation is only responsible for the empty list.

Presentation Issue This thesis presents applied work in formal logics. Hence a lot of formulas, lemmas and theorems show up. The presentation in this thesis follows the motto:

What you see is what we proved!

Isabelle theories can be augmented with LaTeX text which may contain references to Isabelle theorems (by name — see chapter 4 of the Isabelle/HOL tutorial [80]). When Isabelle processes this LaTeX text, it expands these references into the LaTeX text for the proposition of the theorem. Using this mechanism, the text for most of the definitions and theorems in this paper is automatically generated, and hence the chance of typos or omissions is minimised.

The style for the presentation of theorems may also be configured. The plain configuration yields Isabelle's meta logic, e.g.

$$[\![A; B]\!] \Longrightarrow C.$$

This can also be presented as an inference rule:

$$\frac{A \qquad B}{C}$$

To improve readability the theorems may also be schematically converted to a sentence:

If A and B then C.

Or even be filled with user defined text:

Provided A and also B, then we have C.

Most theorems in this thesis are presented in either of the latter two styles.

Although most of the proofs are in the quasi-readable form of Isar proofs, it appears beyond the state of the art to turn these into concise textbook-style proofs automatically. Hence the proofs presented in this thesis are manually tuned variants of the Isar proofs.

Most of the proofs are inductive. Induction on a data type, rule induction on an inductive definition, well-founded induction, or "induction on the recursion-scheme of function …". What does the latter mean? HOL is a logic of total functions. The termination of every recursive definition has to be proven. For primitive recursion on data types the data type package already provides a recursion operator. When defining a general recursive function this is reduced to well-founded recursion and the corresponding proof obligations have to be discharged by Isabelle or manually by the user. Isabelle then derives an induction scheme that exactly follows the recursion in the definition. This induction scheme is often very convenient to use if one attempts to prove a theorem that involves the recursive definition. This is what I refer to with "induction on the recursion-scheme of function …".

CHAPTER 2

The Simpl language

Everything should be made as simple as possible,
but not simpler.
— Albert Einstein

This chapter introduces a general language model for sequential imperative programs. It is independent of a concrete programming language but expressive enough to cover all common language features.

Contents

2.1 Abstract Syntax

In this section I introduce the abstract syntax of Simpl, a ***s**equential* ***im**perative* ***p**rogramming* ***l**anguage*. Simpl is a rather general language that is not fixed to a specific real programming language like C, Pascal or Java. It is more like a model for imperative programs that allows to embed real programming languages for the purpose of program verification. To achieve this flexibility Simpl only defines the control flow statements, without restricting the basic actions or expressions of the programming language. Semantically spoken, Simpl makes no assumptions on the state space, it is polymorphic. We use the canonical type variable $'s$ for the state space. Basic actions are arbitrary state updates of type $'s \Rightarrow 's$. For example we can model variable assignments, field assignments or memory allocation as those basic actions. Simpl is rich enough to cover the following programming language features:

- mutually recursive procedures,
- global variables, local variables and heap,
- runtime faults like array bound violations or dereferencing `null`-pointers
- abrupt termination like `break`, `continue`, `return` and even exceptions,
- side effecting expressions,
- nondeterminism,
- pointers to procedures, partial application and closures, and
- dynamic method invocation.

Despite the fact that Simpl supports nondeterminism, I call it sequential since it has no direct notion of concurrency.

Definition 2.1 ► *Syntax of Simpl*

The syntax of Simpl is defined by the polymorphic data type $('s, 'p, 'f)$ *com, where* $'s$ *is the state space type,* $'p$ *the type of procedure names and* $'f$ *the type of faults. The constructors are listed in the following table, where* c, c_1 *and* c_2 *are Simpl commands of type* $('s, 'p, 'f)$ *com and the Boolean condition* b *and the guard* g *are modelled as state sets of type* $'s$ *set.*

Skip	*Do nothing*
Basic f	*Basic command, where f is a state-update:* $'s \Rightarrow 's$
Seq c_1 c_2	*Sequential composition, also written as* $c_1; c_2$
Cond b c_1 c_2	*Conditional statement (if-then-else)*
While b c	*Loop*
Call p	*Procedure call, p is of type* $'p$
Guard f g c	*Guarded command, fault f is of type* $'f$
Throw	*Initiate abrupt termination*
Catch c_1 c_2	*Handle abrupt termination*
Spec r	*Basic (nondeterministic) command defined by the relation r of type* $('s \times 's)$ *set*
DynCom c_s	*Dynamic (state dependent) command, where* c_s *has type* $'s \Rightarrow ('s,'p,'f)$ *com.*

The formal semantics of Simpl is defined in the next section. Therefore I only give some brief explanations of the language constructs here.

Runtime faults are modelled by the guarded command *Guard f g c*, where g is the guard that watches for runtime faults that may occur in c. If g is violated the fault f is signalled and the computation stops.

The core procedure call *Call p* is parameterless. Parameter passing is implemented in section 2.4.

Throw and *Catch* are the basic building blocks to model various kinds of abrupt termination, like `break`, `continue`, `return` and exceptions. In *Catch* c_1 c_2 the command c_2 can be seen as handler. It is only executed if c_1 terminates abruptly.

Command *Spec r* defines the potential next states via the relation r. It can be used to model nondeterministic choice or in the context of refinement [72, 28] it can represent a specification of a piece of code rather than an implementation.

The dynamic command *DynCom* c_s allows to abstract a command over the state space. It can be used for different purposes: To implement scoping, parameter passing, expressions with side-effects or "real" dynamic construct like pointers to procedures or dynamic method invocations. Details follow in section 2.4.

The set of Simpl commands is not minimal. A *Skip* can be implemented by *Basic* ($\lambda s.\ s$), the dynamic command *DynCom* can be used to implement the conditional, and the *Spec* command can be used to implement the *Basic* command. This separation reflects the different concepts behind the commands.

2.2 Semantics

This section defines the semantics of Simpl by an operational big-step semantics. The core state space is polymorphic and is denoted by the type variable $'s$, runtime faults have type $'f$. To define the semantics the state space is augmented with control flow information:

◂ Definition 2.2 *Extended state space*

$$\textbf{datatype}\ ('s,'f)\ \mathit{xstate} = \mathit{Normal}\ 's \mid \mathit{Abrupt}\ 's \mid \mathit{Fault}\ 'f \mid \mathit{Stuck}$$

Execution starts in a normal state *Normal s*. If a *Throw* is executed the state switches to *Abrupt s*. In case a guard *Guard f g c* is violated the runtime fault is signalled by the state *Fault f*. Moreover, execution can get stuck because of a procedure call to an undefined procedure or an empty set of possible next states in command *Spec*. State *Stuck* makes those dead ends visible in the semantics.

◂ Definition 2.3 *State discriminators*

We introduce the state-discriminators isAbr and isFault:

$$\begin{aligned} &\mathit{isAbr} :: ('s,'f)\ \mathit{xstate} \Rightarrow \mathit{bool}\\ &\mathit{isAbr}\ t \ \equiv\ \exists s.\ t = \mathit{Abrupt}\ s\\[1em] &\mathit{isFault} :: ('s,'f)\ \mathit{xstate} \Rightarrow \mathit{bool}\\ &\mathit{isFault}\ t \ \equiv\ \exists f.\ t = \mathit{Fault}\ f \end{aligned}$$

◂ Definition 2.4 *Big-step semantics of Simpl*

The operational big-step semantics: $\Gamma\vdash \langle c,s\rangle \Rightarrow t$, *is defined inductively by the rules in Figure 2.1. In procedure environment* Γ *execution of command* c *transforms the initial state* s *to the final state* t, *where:*

$$\begin{aligned} &\Gamma\ :: 'p \rightharpoonup ('s, 'p, 'f)\ \mathit{com}\\ &s,t :: ('s, 'f)\ \mathit{xstate}\\ &c\ :: ('s, 'p, 'f)\ \mathit{com} \end{aligned}$$

$$\frac{}{\Gamma\vdash \langle \mathit{Skip},\mathit{Normal}\ s\rangle \Rightarrow \mathit{Normal}\ s}\ (\textsc{Skip}) \qquad \frac{}{\Gamma\vdash \langle \mathit{Basic}\ f,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Normal}\ (f\ s)}\ (\textsc{Basic})$$

$$\frac{\Gamma\vdash \langle c_1,\mathit{Normal}\ s\rangle \Rightarrow s' \qquad \Gamma\vdash \langle c_2,s'\rangle \Rightarrow t}{\Gamma\vdash \langle \mathit{Seq}\ c_1\ c_2,\mathit{Normal}\ s\rangle \Rightarrow t}\ (\textsc{Seq})$$

$$\frac{s \in b \qquad \Gamma\vdash \langle c_1,\mathit{Normal}\ s\rangle \Rightarrow t}{\Gamma\vdash \langle \mathit{Cond}\ b\ c_1\ c_2,\mathit{Normal}\ s\rangle \Rightarrow t}\ (\textsc{CondTrue}) \qquad \frac{s \notin b \qquad \Gamma\vdash \langle c_2,\mathit{Normal}\ s\rangle \Rightarrow t}{\Gamma\vdash \langle \mathit{Cond}\ b\ c_1\ c_2,\mathit{Normal}\ s\rangle \Rightarrow t}\ (\textsc{CondFalse})$$

$$\frac{s \in b \qquad \Gamma\vdash \langle c,\mathit{Normal}\ s\rangle \Rightarrow s' \qquad \Gamma\vdash \langle \mathit{While}\ b\ c,s'\rangle \Rightarrow t}{\Gamma\vdash \langle \mathit{While}\ b\ c,\mathit{Normal}\ s\rangle \Rightarrow t}\ (\textsc{WhileTrue})$$

$$\frac{s \notin b}{\Gamma\vdash \langle \mathit{While}\ b\ c,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Normal}\ s}\ (\textsc{WhileFalse})$$

$$\frac{\Gamma\ p = \lfloor \mathit{bdy}\rfloor \qquad \Gamma\vdash \langle \mathit{bdy},\mathit{Normal}\ s\rangle \Rightarrow t}{\Gamma\vdash \langle \mathit{Call}\ p,\mathit{Normal}\ s\rangle \Rightarrow t}\ (\textsc{Call}) \qquad \frac{\Gamma\ p = \mathit{None}}{\Gamma\vdash \langle \mathit{Call}\ p,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Stuck}}\ (\textsc{CallUndefined})$$

$$\frac{s \in g \qquad \Gamma\vdash \langle c,\mathit{Normal}\ s\rangle \Rightarrow t}{\Gamma\vdash \langle \mathit{Guard}\ f\ g\ c,\mathit{Normal}\ s\rangle \Rightarrow t}\ (\textsc{Guard}) \qquad \frac{s \notin g}{\Gamma\vdash \langle \mathit{Guard}\ f\ g\ c,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Fault}\ f}\ (\textsc{GuardFault})$$

$$\frac{}{\Gamma\vdash \langle \mathit{Throw},\mathit{Normal}\ s\rangle \Rightarrow \mathit{Abrupt}\ s}\ (\textsc{Throw})$$

$$\frac{\Gamma\vdash \langle c_1,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Abrupt}\ s' \qquad \Gamma\vdash \langle c_2,\mathit{Normal}\ s'\rangle \Rightarrow t}{\Gamma\vdash \langle \mathit{Catch}\ c_1\ c_2,\mathit{Normal}\ s\rangle \Rightarrow t}\ (\textsc{Catch}) \qquad \frac{\Gamma\vdash \langle c_1,\mathit{Normal}\ s\rangle \Rightarrow t \qquad \neg\ \mathit{isAbr}\ t}{\Gamma\vdash \langle \mathit{Catch}\ c_1\ c_2,\mathit{Normal}\ s\rangle \Rightarrow t}\ (\textsc{CatchMiss})$$

$$\frac{(s,\, t) \in r}{\Gamma\vdash \langle \mathit{Spec}\ r,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Normal}\ t}\ (\textsc{Spec}) \qquad \frac{\forall t.\ (s,\, t) \notin r}{\Gamma\vdash \langle \mathit{Spec}\ r,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Stuck}}\ (\textsc{SpecStuck})$$

$$\frac{\Gamma\vdash \langle c_s\ s,\mathit{Normal}\ s\rangle \Rightarrow t}{\Gamma\vdash \langle \mathit{DynCom}\ c_s,\mathit{Normal}\ s\rangle \Rightarrow t}\ (\textsc{DynCom})$$

$$\frac{}{\Gamma\vdash \langle c,\mathit{Fault}\ f\rangle \Rightarrow \mathit{Fault}\ f}\ (\textsc{FaultProp}) \qquad \frac{}{\Gamma\vdash \langle c,\mathit{Stuck}\rangle \Rightarrow \mathit{Stuck}}\ (\textsc{StuckProp}) \qquad \frac{}{\Gamma\vdash \langle c,\mathit{Abrupt}\ s\rangle \Rightarrow \mathit{Abrupt}\ s}\ (\textsc{AbruptProp})$$

Figure 2.1: Big-step execution rules for Simpl

The rules for the common language constructs follow the standard textbook semantics [76, 118]. The handling of abrupt termination is adapted from Java exception handling in the semantics of Oheimb [84]. The execution rules come in two flavours: the statement specific rules and the propagation rules for *Fault*, *Stuck* and *Abrupt* states. The statement specific rules are only applicable to *Normal* states. As soon as an "abnormal" state is entered, execution is skipped and the state is propagated. There is no means to recover from a *Fault* or *Stuck* state. Execution ends in those states, whereas the *Catch* statement continues execution of the second statement in a *Normal* state upon abrupt termination of the first statement. If the first statement terminates normally, *Catch* skips the execution of the second one. The encoding of control flow information into the state implements the expected behaviour of abrupt termination. As an example, consider the execution of the statement *Catch* (*Seq Throw* c_1) c_2 starting in state *Normal s*. According to Rules Catch and Seq we first execute $\Gamma\vdash \langle \mathit{Throw},\mathit{Normal}\ s\rangle \Rightarrow \mathit{Abrupt}\ s$. This becomes the first premise of the Seq Rule. The only way to instantiate the second premise is by the AbruptProp Rule: $\Gamma \vdash \langle c_1, Abrupt\ s\rangle \Rightarrow Abrupt\ s$. This means c_1 is skipped and execution of sequential composition becomes $\Gamma \vdash \langle Seq\ Throw\ c_1, Normal\ s\rangle \Rightarrow Abrupt\ s$. According to the Catch Rule, finally the statement c_2 is executed to handle the abrupt termination: $\Gamma \vdash \langle c_2, Normal\ s\rangle \Rightarrow t$.

Branching conditions are modelled semantically as state sets. Therefore testing them becomes set membership. The command *Basic f* applies the function *f* to the current state. Similarly all states that are related to the current state by relation *r* are potential next states of *Spec r*. The semantic becomes nondeterministic here since multiple next states can exist. If there is no possible next state this is signalled by the *Stuck* state. Making those stuck executions and runtime faults visible by the special states *Stuck* and *Fault f*, makes it possible to reason about them in the context of a big-step semantics. Non termination, stuck executions and runtime faults are distinguishable. Only in case of an infinite computation there is no final state defined by this big-step semantics. The dynamic command *DynCom* c_s depends on the current state *s*. The actual command executed is $c_s\ s$.

2.3 Termination

To verify total correctness of a program one needs to show that the program terminates for all valid inputs. To ensure guaranteed termination of a Simpl program it is not sufficient to require the existence of a terminating computation in the big-step semantics: $\exists t.\ \Gamma \vdash \langle c,s\rangle \Rightarrow t$. Due to nondeterminism this does not guarantee that all computations from the same initial state *s* terminate.

◂ Definition 2.5 *Guaranteed termination*

Guaranteed termination: $\Gamma \vdash c \downarrow s$*, of program c in the initial state s is defined inductively by the rules in Figure 2.2, where:*

$$\begin{aligned} \Gamma &:: {'p} \rightharpoonup ({'s}, {'p}, {'f})\ com \\ s &:: ({'s}, {'f})\ xstate \\ c &:: ({'s}, {'p}, {'f})\ com \end{aligned}$$

The rules for guaranteed termination follow the same scheme as the big-step semantics. The statement specific rules only care about *Normal* initial states, whereas programs started in a *Stuck*, *Fault* or *Abrupt* state trivially terminate. Therefore the judgement $\Gamma \vdash c \downarrow s$ only rules out infinite computations, since stuck computations or runtime faults are regarded as terminating. The rules are self-explanatory.

If statement *c* terminates when started in state *s*, then there is a final state *t* with respect to the big-step semantics.

◂ Lemma 2.1

If $\Gamma \vdash c \downarrow s$ *then* $\exists t.\ \Gamma \vdash \langle c,s\rangle \Rightarrow t$.

Proof. By induction on the termination judgement. □

The other direction is not valid, since Simpl is nondeterministic.

2.4 Derived Language Features

The purpose of this section is to illustrate how common programming language features can be modelled in Simpl. For example, we derive procedure calls with parameters from the primitive Simpl statements. In order to give illustrative examples I first introduce an appropriate state space representation.

$$\frac{}{\Gamma\vdash \mathit{Skip} \downarrow \mathit{Normal}\ s}\ (\textsc{Skip}) \qquad \frac{}{\Gamma\vdash \mathit{Basic}\ f \downarrow \mathit{Normal}\ s}\ (\textsc{Basic})$$

$$\frac{\Gamma\vdash c_1 \downarrow \mathit{Normal}\ s \qquad \forall s'.\ \Gamma\vdash \langle c_1,\mathit{Normal}\ s\rangle \Rightarrow s' \longrightarrow \Gamma\vdash c_2 \downarrow s'}{\Gamma\vdash \mathit{Seq}\ c_1\ c_2 \downarrow \mathit{Normal}\ s}\ (\textsc{Seq})$$

$$\frac{s \in b \qquad \Gamma\vdash c_1 \downarrow \mathit{Normal}\ s}{\Gamma\vdash \mathit{Cond}\ b\ c_1\ c_2 \downarrow \mathit{Normal}\ s}\ (\textsc{CondTrue}) \qquad \frac{s \notin b \qquad \Gamma\vdash c_2 \downarrow \mathit{Normal}\ s}{\Gamma\vdash \mathit{Cond}\ b\ c_1\ c_2 \downarrow \mathit{Normal}\ s}\ (\textsc{CondFalse})$$

$$\frac{s \in b \qquad \Gamma\vdash c \downarrow \mathit{Normal}\ s \qquad \forall s'.\ \Gamma\vdash \langle c,\mathit{Normal}\ s\rangle \Rightarrow s' \longrightarrow \Gamma\vdash \mathit{While}\ b\ c \downarrow s'}{\Gamma\vdash \mathit{While}\ b\ c \downarrow \mathit{Normal}\ s}\ (\textsc{WhileTrue})$$

$$\frac{s \notin b}{\Gamma\vdash \mathit{While}\ b\ c \downarrow \mathit{Normal}\ s}\ (\textsc{WhileFalse})$$

$$\frac{\Gamma\ p = \lfloor bdy\rfloor \qquad \Gamma\vdash bdy \downarrow \mathit{Normal}\ s}{\Gamma\vdash \mathit{Call}\ p \downarrow \mathit{Normal}\ s}\ (\textsc{Call}) \qquad \frac{\Gamma\ p = \mathit{None}}{\Gamma\vdash \mathit{Call}\ p \downarrow \mathit{Normal}\ s}\ (\textsc{CallUndefined})$$

$$\frac{s \in g \qquad \Gamma\vdash c \downarrow \mathit{Normal}\ s}{\Gamma\vdash \mathit{Guard}\ f\ g\ c \downarrow \mathit{Normal}\ s}\ (\textsc{Guard}) \qquad \frac{s \notin g}{\Gamma\vdash \mathit{Guard}\ f\ g\ c \downarrow \mathit{Normal}\ s}\ (\textsc{GuardFault})$$

$$\frac{}{\Gamma\vdash \mathit{Throw} \downarrow \mathit{Normal}\ s}\ (\textsc{Throw})$$

$$\frac{\Gamma\vdash c_1 \downarrow \mathit{Normal}\ s \qquad \forall s'.\ \Gamma\vdash \langle c_1,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Abrupt}\ s' \longrightarrow \Gamma\vdash c_2 \downarrow \mathit{Normal}\ s'}{\Gamma\vdash \mathit{Catch}\ c_1\ c_2 \downarrow \mathit{Normal}\ s}\ (\textsc{Catch})$$

$$\frac{}{\Gamma\vdash \mathit{Spec}\ r \downarrow \mathit{Normal}\ s}\ (\textsc{Spec}) \qquad \frac{\Gamma\vdash c_s\ s \downarrow \mathit{Normal}\ s}{\Gamma\vdash \mathit{DynCom}\ c_s \downarrow \mathit{Normal}\ s}\ (\textsc{DynCom})$$

$$\frac{}{\Gamma\vdash c \downarrow \mathit{Fault}\ f}\ (\textsc{Fault}) \qquad \frac{}{\Gamma\vdash c \downarrow \mathit{Stuck}}\ (\textsc{Stuck}) \qquad \frac{}{\Gamma\vdash c \downarrow \mathit{Abrupt}\ s}\ (\textsc{Abrupt})$$

Figure 2.2: Guaranteed termination for Simpl

2.4.1 State Space Representation

To find an adequate model for the representation of the program's state space is a delicate issue. On the one hand it has to be detailed enough to express the properties to be verified, on the other hand it has a decisive impact on the usability of the formal tool. A fine grained model may introduce a lot of "formal noise", which conceals the interesting problems and makes interactive verification a real burden. Moreover, the specific properties of different programming languages can be reflected in the state space model. For example, Java itself ensures that we can only access initialised memory and variables. Whereas in C there are no such guarantees. There it is a desirable requirement of formal verification to ensure the absence of illegal memory accesses. So for Java a state space model that abstracts from initialisation problems is preferable, whereas such a model is out of question for C.

This short discussions makes obvious that we cannot expect one general solution that fits best to all possible applications. This is one of the reasons why Simpl is not commited to one single solution. The state space is not fixed, but polymorphic, so that the user is able to tailor it to his needs. Nevertheless I briefly discuss the solutions that have previously been implemented in HOL and introduce the approach that is used for the examples throughout this thesis. Keep in mind that our goal is to facilitate program verification. We want to support reasoning about individual programs, not reasoning about meta properties of the programming language.

2.4.1.1 State as Function

The first attempt to embed a programming logic into HOL is the work of Gordon [39]. He represents the state as function from names to values: $name \Rightarrow value$. Variable names are first class objects of HOL, which means that we can quantify over them. For example, the property $\forall x.\ x \neq y \longrightarrow s_1\ x = s_2\ x$ expresses that states s_1 and s_2 may only be different for variable y. Such specifications about the *"things that do not change"* are crucial when reasoning about global variables. The range of all variables is represented by the same HOL type, namely *value*. Gordon only considers programs with variables ranging over numbers. Considering variables of different types or even composite types like arrays, a more complex representation of values is needed. In his formalisation of Dijkstra, Harrison [46] uses an inductive data type to handle this problem:

datatype *value* = *Intg int* | *Bool bool* | *Array value list*

The different representations for integers (*int*), Booleans (*bool*) and arrays (*value list*) are injected into the type *value* by the constructors *Intg*, *Bool* and *Array*. By modelling arrays as lists of values also nested arrays can be expressed. An example for an array of array of integers is *Array* [*Array* [*Intg 1, Intg 2*], *Array* [*Intg 3, Intg 4*]]. A drawback of this approach is that type constraints that are usually imposed by the programming language can easily be violated. A mixed array like *Array* [*Intg 1, Bool b*] is a perfectly legal *value* but typically ruled out by the type system of the programming language. This problem carries on to expressions, where we have to explicitly deal with programming language typing within HOL. Consider the simple statement `x := y + 1`. Such an assignment boils down to a function update in our state space. To handle the addition we somehow have to lift the HOL addition that is defined for type *int* to type *value*. There are two solutions. We can either **project the arguments** or **lift the operation**.

Projecting arguments (aggressive evaluation): We define a function *the-Intg* that projects an *int* out of a *value*:

$$\begin{array}{l} \textit{the-Intg} :: \textit{value} \Rightarrow \textit{int} \\ \textit{the-Intg}\ (\textit{Intg}\ i) = i \end{array}$$

Since HOL is a logic of total functions the term *the-Intg* (*Bool b*) is legal, but results in an unspecified *int*. The assignment `x := y + 1` is then semantically modelled as:

$$s(x := \textit{Intg}\ (\textit{the-Intg}\ (s\ y) + 1)).$$

Here s is the current state, and := means function update in HOL. We have to insert projections and injections into the original expression, which carries on to assertions about the program and therefore clutters up the verification task.

Lifting operations (type-sensitive evaluation): In this approach we explicitly fix the binary operations and define their evaluation for *value*s:

$$\textbf{datatype}\ bop = Add \mid And$$

$$\begin{aligned} &eval :: (bop \times value \times value) \Rightarrow value\\ &eval\ (Add, Intg\ n, Intg\ m) = Intg\ (n + m)\\ &eval\ (And, Bool\ b, Bool\ c)\ \ = Bool\ (b \wedge c)\end{aligned}$$

Again *eval* is under-specified, if the arguments have different types. In this setting our assignment `x := y + 1` becomes:

$$s(x := eval\ (Add, s\ y, Intg\ 1)).$$

Since the set of possible operations is made explicit by the data type *bop*, the evaluation function *eval* can take care of typing issues and implicitly perform the projections from *value*. However, now primitive values like *1* have to be injected into type *value*. Moreover, basic properties of the operations only hold for correctly typed expressions. For example, commutation of addition: $eval\ (Add, n, m) = eval\ (Add, m, n)$ only holds, if we know that both arguments are of the form *Intg i*. In this case we can reduce the addition on type *value* to the ordinary integer addition and inherit their properties. We need to insert those explicit type constraints into the assertions about the program to be able to lift the logical properties of the operations for types *int* or *bool* to type *value*. This basically means that we prove type safety of evaluation every time we reason about expressions. This is annoying, since for a type-safe programming language this can be proven once and for all.

To summarise, the characteristics of the "state as function" approach are the following:

- only fixed range of variables and
- explicit typing, but
- first class variable names.

2.4.1.2 State as Tuple

As alternative Wright *et al.* [111] propose to take a tuple to represent the state space. Variables are identified by their position in the tuple rather than by names. For example, the tuple $int \times int \times bool$ represents a state space with three variables of type *int* or *bool*, respectively. Each variable thereby has an individual HOL type. The typing issues of the "state as function" approach are eliminated since we do not have to introduce an artificial super-type for all variables. Variable types are identified with HOL types and thus type inference only accepts welltyped expressions.

By choosing the names of bound variables, when abstracting over the state space, one can even name the programming language variables. Abstraction naturally occurs in assertions, if they are represented as predicates $state \Rightarrow bool$, or in update functions $state \Rightarrow state$. For example, the state update of our running example `x := y + 1` can be encoded in the following function:

$$\lambda(x, y, b).\ (y + 1, y, b).$$

Via the λ-abstraction the components of the tuple are named with x, y and b. If all variables are known this translation from the assignment to the state update can be handled by a mere syntactic translation. However, great care has to be taken, since those translations have to remember all the variable names and their order in the tuple. Moreover, names of bound variables can only be considered as comments for the reader. Logically there is no difference between $\lambda(x, y, b).\ (y + 1, y, b)$ and $\lambda(n, m, k).\ (m + 1, m, k)$. Besides, an one-to-one translation between the input and output syntax is impossible in some cases. Consider the two assignments `x := x` and `y := y`. Both would be mapped to the same internal form: $\lambda(x, y, b).\ (x, y, b)$.

Since variables do not have a real name, we cannot quantify over variable names. Fortunately the "typical" assertions do not quantify over variable names anyway. They just refer to the components of the state. This can be done in the same fashion as the state update above. However, how can we express the *"things that do not change"*? To express that only `y` may change its value, one can list all components that do not change: $x_1 = x_2 \wedge b_1 = b_2$. So in principle it is possible to express it, but the major drawback is the poor modularity. Every time we add a new variable to the program, we have to adapt those specifications. The problem is that the way to express that `y` may not change, does not even mention `y` at all, but instead explicitly lists all the other variables. Similarly, all assertions and functions on the state have to be updated if a variable is added, since they split the tuple to its components.

We end up with the following characterisation of the "state as tuples" approach:

- variables can range over any HOL type,
- automatic typing by type inference, but
- variables have only fake names.

2.4.1.3 State as Record

Records are similar to tuples, but additionally allow us to give proper names to variables. They were proposed by Wenzel [115] as state space representation and successfully used by Prensa [98] for the verification of parallel programs. Records enhance tuples by supplying selection and update functions for each component. For example

record *state* =
x :: *int*
y :: *int*
b :: *bool*

yields the selectors $x :: state \Rightarrow int$, $y :: state \Rightarrow int$ and $b :: state \Rightarrow bool$, and the update functions $x\text{-}update :: int \Rightarrow state \Rightarrow state$, $y\text{-}update :: int \Rightarrow state \Rightarrow state$ and $b\text{-}update :: bool \Rightarrow state \Rightarrow state$. A record update $x\text{-}update\ i\ s$ can be abbreviated with $s(\!|x := i|\!)$. In this setting the assignment `x := y + 1` becomes a record update:

$$s(\!|x := y\ s + 1|\!).$$

As with tuples we still cannot quantify over variable names, since record field names are no first class objects of HOL. A field is merely characterised by its selection and update functions. However, we can now specify that only y may change, without having to mention the other variables: $\exists i.\ s_2 = s_1(\!|y := i|\!)$. So in both this specification and the assignment above, only the relevant portions of the state space occur. This improves modularity compared to tuples.

In the end I decided to use records as state space representation. Every variable is represented by a record field and thus has an individual HOL type. As with tuples, automatic type inference takes care of typing issues. Moreover, with field selection and update we have convenient means to express state updates and assertions.

As final remark, there is one oddity of the "state as record" approach. Modelling the state as function gives us a uniform representation of the state space for every program. In the other approaches the shape and therefore the type of the state depends on the variables of each individual program. Each variable needs an extra slot in the tuple or field in the record. However, as sketched above, with records we can focus on the relevant variables in the assertions and state updates. So for practical issues this is no problem. In Sections 6 and 9.2 this discussion is continued.

2.4.2 Concrete Syntax for Simpl

To improve readability of Simpl programs I introduce pseudo-code syntax. First of all let me address the assignment. With the state represented as record, an assignment `m = m - 1` is mapped to a *Basic* command that performs a record update: *Basic* $(\lambda s.\ s(\!|m := m\ s - 1|\!))$. The record update and the record selection both refer to the program state s that is bound by the λ. Whenever we refer to a component of the state space it is type-set in a sans-serif font. In Isabelle those components are marked with the acute symbol ´. So m is typeset as ´m in Isabelle and expands to $m\ s$ for some bound state s. The abstraction over this state is triggered by the statements, like the function in *Basic* or the condition of *Cond* and *While*. Moreover we introduce the special braces $\{\!|\ldots|\!\}$ to describe sets that implicitly abstract over the state. The following table lists concrete syntax and its mapping to the Simpl commands for some basic statements.

concrete syntax	abstract syntax
SKIP	*Skip*
m := e	*Basic* $(\lambda s.\ s(\!|m := e|\!))$
c_1; c_2	*Seq* $c_1\ c_2$
IF b **THEN** c_1 **ELSE** c_2 **FI**	*Cond* $\{s.\ b\}\ c_1\ c_2$
IF b **THEN** c_1 **FI**	*Cond* $\{s.\ b\}\ c_1$ *Skip*
WHILE b **DO** c **OD**	*While* $\{s.\ b\}\ c$
TRY c_1 **CATCH** c_2 **END**	*Catch* $c_1\ c_2$
$g \mapsto c$	*Guard False* $g\ c$
$\{\!|P|\!\}$	$\{s.\ P\}$

By default, faults $'f$ are instantiated with Boolean values. The guarded statement $g \mapsto c$ is marked with fault *False*. In most cases we want to prove that all guards hold and thus no fault at all occurs. Hence the fault, which guards are marked with, is not important. However, in Chapter 5 we introduce an interface to discharge guards with automatic program analysis. There, a default marking of guards with *False* comes in handy.

2.4.3 Expressions with Side Effects

Simpl has no built in expression language. Expressions appear inside a statement as ordinary HOL expressions. For example, the assignment `m = m - 1` is mapped to a *Basic* command where the subtraction appears inside the state update function: $Basic\ (\lambda s.\ s(\!|m := m\ s - 1|\!))$. Therefore expressions do not have side-effects. To deal with side-effecting expressions of a programming language, the trivial approach is to reduce them to statements and expressions without side effects. A program transformation step introduces temporary variables to store the result of subexpressions. For example, we can get rid of the increment expression in `r = m++ + n` by first saving the initial value of `m` in a temporary variable: `tmp = m; m = m + 1; r = tmp + n`. In our state space model this approach is somehow annoying since the temporary variables directly affect the shape of the state record. The essence of the temporary variables is to fix the value of an expression at a certain program state, so that we can later on refer to this value. Since our dynamic command *DynCom* allows to abstract over the state space we already have the means to refer to certain program states. Similar to the state monad in functional programming [112] we introduce the command *bind e c*, which binds the value of expression *e* (of type $'s \Rightarrow 'v$) at the current program state and feeds it into the following command *c* (which is of type $'v \Rightarrow ('s, 'f, 'p)\ com$):

$$bind :: ('s \Rightarrow 'v) \Rightarrow ('v \Rightarrow ('s,'f,'p)\ com) \Rightarrow ('s,'f,'p)\ com$$
$$bind\ e\ c \equiv DynCom\ (\lambda s.\ c\ (e\ s))$$

◂ Definition 2.6

We introduce the notation $e \gg m.\ c$ as syntactic sugar for $bind\ e\ (\lambda m.\ c)$. The assignment `r = m++ + n` is represented in Simpl as:

$$\mathsf{m} \gg m.\ \mathsf{m} := \mathsf{m} + 1;\ \mathsf{r} := m + \mathsf{n}.$$

Unfolding the definition of bind we arrive at:

$$DynCom\ (\lambda s.\ \mathsf{m} := \mathsf{m} + 1;\ \mathsf{r} := m\ s + \mathsf{n}).$$

The last occurance of *m* refers to the initial state *s*.

As the intermediate names, introduced by a *bind*, are only bound names of a λ-abstraction it is possible to supply more syntactic sugar to completely hide the names and mimic the original increment expression of C.

2.4.4 Abrupt Termination

Abrupt termination is the immediate transfer of control flow to some *enclosing* statement, skipping the execution of the pending statements that normally would be processed. The enclosing statement can be syntactically determined like in case of `break`, `continue` and `return` or dynamically like a handler for exceptions. Abrupt termination is well-behaved compared to arbitrary gotos. We can only jump out, but not inside or criss-cross the code. The building block for abrupt termination in Simpl is:

$$\textbf{TRY}\ c_1\ \textbf{CATCH}\ c_2\ \textbf{END}.$$

Abrupt termination of c_1 is handled by c_2. In case of normal termination of c_1, the second statement c_2 is skipped.

To `break` out of a loop means to immediately exit the loop. This can be implemented by putting the loop between the **TRY**−**CATCH**:

TRY WHILE b **DO** … **THROW** … **OD CATCH SKIP END**.

In case of `continue` only the loop body is exited and control flow continues with a new iteration of the loop. Therefore only the loop body is protected:

WHILE b **DO TRY** … **THROW** … **CATCH SKIP END OD**.

Similarly, in case of a `return`, the procedure body is protected by the enclosing **TRY**−**CATCH**. Exception handling can be directly mapped to **TRY** c_1 **CATCH** c_2 **END**. The protected area is c_1 and the exception handler is c_2.

In case all kinds of abrupt termination are simultaneously present, we instrument the state space to distinguish them. The auxiliary variable Abr stores the reason for abrupt termination. Then `break`, for example, actually becomes Abr := *''break''*; **THROW**. The corresponding handler peeks into Abr to decide whether to stop abrupt termination or to continue it: **IF** Abr = *''break''* **THEN SKIP ELSE THROW FI**. Similarly, exception objects can be stored in an auxiliary variable, so that the handler can make its decision to catch or re-raise the exception. We only have to use a global variable to ensure that the exception properly passes procedure boundaries.

2.4.5 Blocks and Procedures with Parameters

The purpose of blocks in Simpl is to implement scoping. They can be used to introduce local variables and to handle parameter passing in procedures. Again we use the state abstraction provided by *DynCom* to get hold of the initial state. This way we can restore the contents of the initial state when we exit the block.

Definition 2.7 ► *block*

block :: $('s \Rightarrow 's) \Rightarrow ('s,'p,'f)\ com \Rightarrow ('s \Rightarrow 's \Rightarrow 's) \Rightarrow ('s \Rightarrow 's \Rightarrow ('s,'p,'f)\ com) \Rightarrow ('s,'p,'f)\ com$
block init bdy return c ≡
DynCom
 (*λs.* **TRY** *Basic init*; *bdy* **CATCH** *Basic* (*return s*); **THROW END**;
  DynCom (*λt. Basic* (*return s*); *c s t*))

A procedure call with parameters can directly be implemented as a block with the parameterless call as body:

Definition 2.8 ► *Procedure call with parameters*

call :: $('s \Rightarrow 's) \Rightarrow 'p \Rightarrow ('s \Rightarrow 's \Rightarrow 's) \Rightarrow ('s \Rightarrow 's \Rightarrow ('s,'p,'f)\ com) \Rightarrow ('s,'p,'f)\ com$
call init p return c ≡ *block init* (*Call p*) *return c*

The control flow of statement *block init bdy return c* is illustrated in Figure 2.3. First

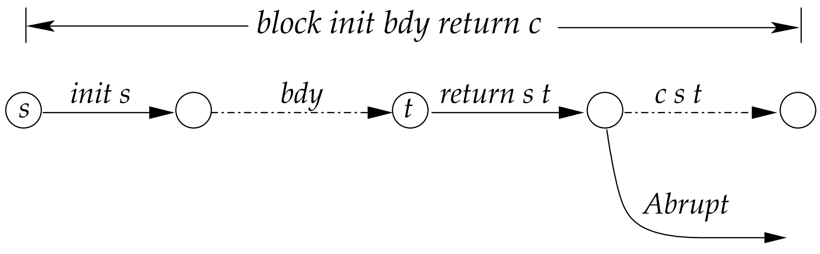

Figure 2.3: Control flow of a *block*

the function *init* initialises the block before body *bdy* is executed. Function *return* exits the block. To communicate results to the enclosing environment the following statement c can peek inside the block, similar to the *bind* command. This can be used to pass the result of a procedure to the caller. The block also takes care about abrupt termination. For both abrupt and normal termination of the block body the state is cleaned up by function *return*. In case of abrupt termination the follow-up statement c is skipped and abrupt termination is propagated by re-raising **THROW**. The initial state before entering the body is captured by the first *DynCom* and is named s. The final state of the body is bound to t by the second *DynCom*. The *return* function as well as the follow-up statement c can refer to both states. This allows *return* to restore the initial values of hidden variables and the follow-up statement c can implement any kind of evaluation strategy for procedure calls. Consider a procedure call like `p -> next = reverse (p)` in C. The left-value of `p -> next` determines the address where the return value of `reverse` is stored. According to the left to right evaluation strategy this address is calculated before the procedure call. Since the procedure itself can modify the global state due to side-effects the left-value of `p -> next` may evaluate to a different address after the procedure call. To properly model the final assignment, we have to restore the initial address of left-value `p -> next` after the procedure returns. For this purpose we can access the initial state s in the follow-up statement c.

The following examples illustrate the Simpl mechanisms for scoping by blocks and procedures. Introducing a new local variable as in {`int i = j; bdy`} can be implemented by:

$$block\ (\lambda s.\ s(\!|i := j\ s|\!))\ bdy\ (\lambda s\ t.\ t(\!|i := i\ s|\!))\ (\lambda s\ t.\ \textbf{SKIP}).$$

The block is entered by initialising variable i with the current value of j. To exit the block the initial value of i is restored. Since no result value is passed the follow-up statement is just **SKIP**.

Parameter passing for procedure calls is concerned with formal and actual parameters. Suppose that the formal parameter of procedure `foo` is `n`. Parameter passing for `foo(i)` means to copy the content of `i` to `n`:

$$init = (\lambda s.\ s(\!|n := i\ s|\!)).$$

To return from the procedure the local variables of the caller have to be restored. Or put the other way round, only the global parts of the state are propagated from the procedure to the caller. We group together all global variables in another record *globals*, which is a field of the state space record. Hence the return from a procedure can be expressed as follows:

$$return = (\lambda s\ t.\ s(\!|globals := globals\ t|\!)).$$

Pascal [54] also allows nested local procedure definitions. The local variables of an enclosing procedure act as global variables in the local procedures. Hence a uniform distinction between global and local variables is no longer adequate. However, as Pascal supports static scoping we know at every call point of a procedure which variables are regarded as global and which as local. This can also be encoded into the *return* function.

The final question is how the result of a function call can be communicated to the caller. A statement like `return e` in C has two consequences: The procedure is abruptly terminated and the value of `e` is passed to the caller. In Simpl we decompose both aspects. We keep an auxiliary variable *res* where the result is stored. So `return` e basically becomes res := *e*. If we also want to model the abrupt termination we can add a subsequent **THROW** (cf. Section 2.4.4). Since the result is stored in *res* the follow-up statement can get it from there. The assignment `j = foo(i)` can be encoded in the follow-up statement:

$$c = (\lambda s\ t.\ \mathit{Basic}\ (\lambda u.\ u(\!|j := \mathit{res}\ t|\!))).$$

Note that *s* is the initial state of the caller, *t* the final state of the procedure body and *u* the current state after returning from the procedure. To summarise, the procedure call `j = foo(i)` is modelled by:

$$\mathit{call}\ (\lambda s.\ s(\!|n := i\ s|\!))\ \text{"foo"}\ (\lambda s\ t.\ s(\!|\mathit{globals} := \mathit{globals}\ t|\!))\ (\lambda s\ t.\ \mathit{Basic}\ (\lambda u.\ u(\!|j := \mathit{res}\ t|\!))).$$

2.4.6 Dynamic Procedure Calls

A dynamic procedure call is a combination of the dynamic command *DynCom* and a procedure call with parameters.

Definition 2.9 ► *Dynmaic procedure call*

$\mathit{dynCall}{::}('s{\Rightarrow}'s) \Rightarrow ('s{\Rightarrow}'p) \Rightarrow ('s{\Rightarrow}'s{\Rightarrow}'s) \Rightarrow ('s{\Rightarrow}'s{\Rightarrow}('s,'p,'f)\ \mathit{com}) \Rightarrow ('s,'p,'f)\ \mathit{com}$
$\mathit{dynCall}\ \mathit{init}\ p\ \mathit{return}\ c \equiv \mathit{DynCom}\ (\lambda s.\ \mathit{call}\ \mathit{init}\ (p\ s)\ \mathit{return}\ c)$

The procedure name can depend on the current state. Hence the procedure that actually gets called can depend on the state, like on the value of a variable. We can model a pointer to a procedure that way. A procedure pointer is a variable that stores the name of the procedure. Moreover, in an object oriented setting we can model dynamic method invocation. The method called depends on the dynamic type of the object. This dynamic type is obtained from the current state.

2.4.7 Closures

The concept of closures is used in functional programming to handle partial application of function. A function is a first class value in functional programming languages and hence can be passed like any other value as a parameter to other functions. This allows to implement higher order functions. If a function application only provides a function with a part of its parameters, the function cannot yet be evaluated. The computation is postponed until all parameters are supplied. The parameters of the partial application are stored together with the function to make them available as the function finally gets evaluated. The combination of a local parameter environment and the function is the so called closure.

A similar effect occurs in imperative languages like Algol 60 [30] or Pascal [54] which allow procedures as parameters of procedures (similar to procedure pointers) and also local procedure declarations and static scoping. A local procedure *L* can be declared in the (static) scope of an enclosing procedure *E*. The local procedure *L* can access the local variables of the enclosing procedure *E*. However, the local procedure can escape from the scope of the enclosing procedure *E* if is passed as an argument to an other procedure *P*. If the local procedure *L* is finally called in the body of procedure *P* it has to remember the original local variables of the statically enclosing procedure *E*. As long as the locally declared procedure *L* only *reads* from the local variables of the enclosing procedure *E*, this can also be expressed by partial application. We can move the local procedure declaration to the outermost (global) scope by augmenting its parameter list with all the local variables of the enclosing procedure E. Let L' be this global version of the procedure L. The local procedure declaration L is then a partial application of the corresponding global procedure L' to the local variables of the enclosing procedure E.

In Simpl we represent closures as pairs $'e \times 'p$, where $'e$ is the environment and $'p$ the procedure name. To call a closure we first adapt the current state according to the environment and then call the procedure. The function *upd* takes the environment and transforms it to a state update $'s \Rightarrow 's$, like the *init* function in procedure calls.

◄ Definition 2.10

$$\begin{array}{l} \mathit{callClosure} :: ('e \Rightarrow ('s \Rightarrow 's)) \Rightarrow ('e \times 'p) \Rightarrow ('s,'p,'f)\ \mathit{com} \\ \mathit{callClosure}\ \mathit{upd}\ \mathit{cl} \equiv \mathit{Basic}\ (\mathit{upd}\ (\mathit{fst}\ \mathit{cl}));\ \mathit{Call}\ (\mathit{snd}\ \mathit{cl}) \end{array}$$

This command only allows to call a fixed closure. We use it to specify the expected behaviour of a closure. In a program we actually want to retrieve the closure from the state, as a generalised procedure pointer, and then call it including additional parameters and result passing. We can define it analogously to the dynamic procedure call. The state update resulting from the environment in the closure is just composed to the *init* function that applies the remaining parameters.

◄ Definition 2.11
Calling a closure

$$\begin{array}{l} \mathit{dynCallClosure} :: ('s \Rightarrow 's) \Rightarrow ('e \Rightarrow ('s \Rightarrow 's)) \Rightarrow ('s \Rightarrow ('e \times 'p)) \Rightarrow \\ ('s \Rightarrow 's \Rightarrow 's) \Rightarrow ('s \Rightarrow 's \Rightarrow ('s,'p,'f)\ \mathit{com}) \Rightarrow ('s,'p,'f)\ \mathit{com} \\ \mathit{dynCallClosure}\ \mathit{init}\ \mathit{upd}\ \mathit{cl}\ \mathit{return}\ c \equiv \\ \mathit{DynCom}\ (\lambda s.\ \mathit{call}\ (\mathit{upd}\ (\mathit{fst}\ (\mathit{cl}\ s)) \circ \mathit{init})\ (\mathit{snd}\ (\mathit{cl}\ s))\ \mathit{return}\ c) \end{array}$$

Let us have a look at an example. We represent the local environment in a closure as a list of pairs, associating the name of a parameter with its value. We have a variable c in the state space that can store a closure. Consider a procedure *"Add"* that takes two parameters x and y and returns the addition in result variable r. To partially apply procedure *"Add"* to a variable n we create a closure and associates *"x"* with the current value of n:

$$\mathsf{c} := ([(''x'', \mathsf{n})], ''Add'').$$

In case *"Add"* would expect more than two parameters, further partial applications would add the additional bindings to the association list. To call the closure we need to define a function that converts the association list to a state update. First, we associate the parameter names to the corresponding update functions of the state space record *st*:

$$\begin{array}{l} \mathit{var} :: \mathit{string} \rightharpoonup (\mathit{nat} \Rightarrow (\mathit{st} \Rightarrow \mathit{st})) \\ \mathit{var} \equiv [''x'' \mapsto x\text{-}\mathit{update},\ ''y'' \mapsto y\text{-}\mathit{update}] \end{array}$$

Then we iterate those update functions over the association list:

$$\begin{array}{l} \mathit{upd} :: (\mathit{string} \times \mathit{nat})\ \mathit{list} \Rightarrow \mathit{st} \Rightarrow \mathit{st} \\ \mathit{upd}\ \mathit{es}\ s \equiv \mathit{foldl}\ (\lambda s\ (x, v).\ \mathit{the}\ (\mathit{var}\ x)\ v\ s)\ s\ \mathit{es} \end{array}$$

Here is an example:

$$\begin{array}{l} \mathsf{n} := 2; \\ \mathsf{c} := ([(''x'', \mathsf{n})], ''Add''); \\ \mathsf{n} := 1; \\ \mathsf{m} := 3; \\ \mathit{dynCallClosure}\ (\lambda s.\ s(\!|y := m\ s|\!))\ \mathit{upd}\ c\ (\lambda s\ t.\ s(\!|\mathit{globals} := \mathit{globals}\ t|\!))\ (\lambda s\ t.\ \mathsf{r} := r\ t). \end{array}$$

The code snippet first partially applies *"Add"* to the current value of n, which is *2*. Then it modifies n and finally calls the closure while supplying the second parameter with the current value of m. Note that *c* is the selector of the state record. The result is stored in variable r. Since the closure has stored the initial value of n the result is *5* and not *4*.

2.4.8 Arrays

Arrays are modelled as HOL lists. The *i*'th element of a list *l* is obtained by $l_{[i]}$. Update of the *i*'th element is $l[i := e]$. An assignment to an array is written as $\mathsf{a}_{[i]} := e$ and translates to $\mathsf{a} := \mathsf{a}[i{:=}e]$.

To check for array bound violations with a guard we can use the length of the list, eg.: $\{\!|\mathsf{i} < |\mathsf{a}||\!\} \mapsto \mathsf{a}_{[\mathsf{i}]} := e$

2.4.9 Heap and Pointers

The heap model we introduce in this section excludes explicit address arithmetic but is capable to represent typical heap structures like lists:

```
struct list {int cont; struct list *next}.
```

I want to emphasise that this is only one possible heap model. As the Simpl language does not restrict the state space it can deal with any kind of heap representation. A proper heap model depends on the level of abstraction the programming language offers as well as the concrete applications we attempt to tackle within the verification environment. Tuch and Klein [110] present an alternative heap model for Simpl that is capable to deal with low-level manipulations like pointer arithmetic in untyped memory, but still offers a neat, abstract and typed view of memory where possible. The heap model we adapt is the split heap approach that goes back to Burstall [20] and was recently taken up by Bornat [17] and also by Metha and Nipkow [67, 68]. The main benefit of this heap model is that it already excludes aliasing between different fields of structures, like `cont` and `next` in the list example. The typed view of memory is hard-wired into the model. Thats why we cannot properly express low-level untyped operations like pointer arithmetic in it.

To highlight that we do not calculate with pointers we introduce a type *ref* of references. We use the *typedef* facility of Isabelle/HOL [80] to construct the new type *ref* that is isomorphic to the natural numbers. *UNIV* is the universal set:

$$\textbf{typedef}\ \mathit{ref} = \mathit{UNIV}{::}\mathit{nat\ set}.$$

The typedef mechanism defines a new type from a subset of an existing type. It also provides functions to convert between both types. By introducing the new type *ref* without lifting the arithmetic operations from the natural numbers, we exclude address arithmetic. We declare the reference *NULL* as a constant without any definition, it is just one value upon the references.

2.4.9.1 Split Heap Approach

In the context of structures in the heap one might naturally think of the heap as a function from addresses to a heap object which contains the structure. So if we attempt to access a component of a heap object we need the pointer and the field name. Hence the heap can also be viewed as a function that takes those two arguments and retrieves an atomic value:

$$heap :: ref \Rightarrow fieldname \Rightarrow val.$$

Updating a heap h at reference p and field f with a value v can be described as the following nested function update:

$$h(p{:=}(h\ p)(f := v)).$$

If we reason about programs with pointers we usually do not reason about concrete reference values. Instead we might have two pointer variables p and q. Due to aliasing those pointers may reference the same location. Hence we cannot infer from different variable names that they point to different locations. With the field names this is different. These are constants that are fixed by the types that occur in the program. If two field-names are different the locations they address are also different. Moreover the field-names are also constant in the program text. For example, for dereferencing a pointer `x -> next` the field-name `next` is a literal. It is no variable. This is the key ingredient that is exploited in the split heap approach. First it just swaps the arguments of the heap:

$$heap :: fieldname \Rightarrow ref \Rightarrow val.$$

The above heap update of field f and reference p now becomes:

$$h(f := (h\ f)(p := v)).$$

Any attempt to access a different field g is already handled by the outer function update instead of the inner one in the previous example. Since the field names are fixed by the program we can merge the heap with each field-name and arrive at the split heap model. This means that each field of a structure gets a separate heap in the state space. In the list example we introduce the heaps $cont :: ref \Rightarrow int$ and $next :: ref \Rightarrow ref$. As additional benefit the fields can have individual HOL types, like *int* or *ref* and do not have to be injected into a single type *val*. We also introduce heaps for all primitive values we attempt to point to, like a heap for integers or a heap for Booleans. The split heap model excludes aliasing on the granularity of the fields of a structure. Hence it also excludes aliasing between different structures like lists and trees. A *next* pointer of a list never collides with a *left* or *right* pointer of a tree. An update to the *next* heap does not affect the *left* or *right* heaps. Like each variable gets its own component in the state space record, each structure field gets its own component. This is also extended to nested structures.

Here is the layout of the state space record for a heap that can contain lists and trees as well as pointers to integers and Booleans.

record *state* =	**record** *heap* =
$globals :: heap$	$cont :: ref \Rightarrow int$
… <*local variables*> …	$next :: ref \Rightarrow ref$
	$left :: ref \Rightarrow ref$
	$right :: ref \Rightarrow ref$
	$int :: ref \Rightarrow int$
	$bool :: ref \Rightarrow bool$

As already mentioned in Section 2.4.5 all global components like the split heaps are grouped together in a single record field *globals*. This gives a uniform model for the procedure return, regardless of the number of split heaps in the program.

To access the heap we provide the syntax $p \to f$ that mimics dereferencing pointers in C. The syntax $p \to f$ is translated to function application $f\ p$. Moreover, the translation to variables and heap components takes care of the indirection to the *globals* component of the state. For example, the assignment $\mathsf{p} := \mathsf{p} \to \mathsf{next}$ is translated to

$$Basic\ (\lambda s.\ s(\!|p := (next\ (globals\ s))\ p|\!)),$$

where p is considered to be a local variable. Similarly, the heap update $\mathsf{p} \to \mathsf{next} := v$ is translated to

$$Basic\ (\lambda s.\ s(\!|globals := (globals\ s)(\!|next := (next\ (globals\ s))(p\ s := v)|\!)|\!)).$$

2.4.9.2 Memory Management

To model allocation and deallocation we need some bookkeeping of allocated references. This can be achieved by an auxiliary ghost variable alloc in the state space. A good candidate is a list of allocated references. We do not commit ourselves to a certain allocation strategy. We only demand "freshness" of a new reference. We use Hilberts choice operator to select a fresh reference:

Definition 2.12 ▸
$$\begin{aligned}&new :: ref\ set \Rightarrow ref\\ &new\ A \equiv SOME\ a.\ a \notin \{NULL\} \cup A\end{aligned}$$

Since type *ref* is isomorphic to the natural numbers we have infinitely many references. If only finitely many references are allocated we can always find a fresh reference.

Lemma 2.2 ▸ *If finite A* then *new A ∉ A ∧ new A ≠ NULL.*

The global ghost variable alloc is a list of allocated references. Every time the program allocates memory it is augmented with the new reference *new* (*set* alloc). Similarly, if the program deallocates a reference it is removed from the list. The number of elements in a list is *per se* finite. Hence we can always get a new "fresh" reference according to Lemma 2.2.

By the length of the list we can also handle space limitations. If we need a more detailed model for the free memory we can also introduce another ghost variable free that counts the free memory cells. With this slightly more general view the objects can have different sizes. Although we only need one reference to store any object in the split heap model, the allocation of different objects can still have an individual impact on the free counter, depending on the size of the objects.

To guard against dangling pointers we can regard the allocation list:

$$\{\!|\mathsf{p} \neq Null \wedge \mathsf{p} \in set\ \mathsf{alloc}|\!\} \mapsto \mathsf{p} \to \mathsf{cont} := 2.$$

The use of guards is a flexible mechanism to adapt the model to the kind of language we are looking at. In case of a type-safe language like Java there is no explicit deallocation by the user and we can remove some guards. For example, the test for $\mathsf{p} \in set\ \mathsf{alloc}$ is not necessary in Java. If the `new` instruction of the programming language does not initialise the allocated memory we can add another ghost variable to watch for initialised memory through guards.

2.5 Conclusion

Simpl is a mixture of a deep and shallow embedding of a programming language in HOL. The statements are embedded deeply but allow a shallow embedding of basic operations and expressions on the polymorphic state space. On the one hand the deep embedding allows to define functions and predicates by recursion over the statement syntax and supplies an induction principle for statements. On the other hand the shallow parts provide the flexibility to tailor the language to the requirements of a concrete programming language and verification task.

There is no need for a type system on the high level of abstraction that Simpl provides. Moreover, the representation of the state space as a record allows to map primitive types of the programming language to HOL types.

CHAPTER 3

Hoare Logic for Simpl

This chapter describes a Hoare logic for both partial and total correctness of Simpl programs. The soundness and (relative) completeness of the Hoare logic is proven.

Contents

Program verification in general is about the derivation of program properties. For sequential programs the main focus of interest is on functional correctness, absence of runtime faults and guaranteed termination. More fine-grained properties include execution time or memory consumption. To derive those properties one can directly argue about the semantics or use a program logic. A program logic supplies a calculus for reasoning about programs, without directly referring to the semantics. Depending on the intended purpose, a program logic can be specialised to certain program properties, possibly even allowing automated deduction, or giving the user a more abstract or convenient methodology for reasoning. The essence of imperative programming is to manipulate the program state. The most prominent program logics for imperative programs is Hoare logic, developed by Hoare [47] on the basis of earlier work by Floyd [35]. The basic idea is to describe the properties of all possible states at a given program point by an assertion, expressed as a logical formula. The Hoare logic gives rules for every programming language construct, which describe the effect of the program directly on the assertion instead of the state. Reasoning about a program is lifted to the level of assertions. An annotated program can be completely reduced to the assertion logic, for which first order logic is sufficient. Higher order concepts like induction are no longer necessary to reason about loops, if a proper invariant is supplied. This is the main theoretical and practical benefit of Hoare logics compared to direct reasoning on the program semantics.

A Hoare logic judgement is usually of the form $\{P\}\ c\ \{Q\}$, where P and Q are assertions namely the pre- and the postcondition and c is a program fragment. If a state satisfies the precondition P then the final state after execution of c satisfies the postcondition Q. If termination is also guaranteed one speaks about total correctness, otherwise about partial correctness. There is no uniform treatment of runtime-faults, like division by zero, in the literature. Sometimes they are regarded as non-terminating computations and thus have to be proven absent to ensure total correctness, sometimes they are simply ignored. Some programming languages like Java use exceptions and thus introduce means to handle runtime faults within the programming language. Of course a proper Hoare logic then has to deal with exceptions.

Traditionally, assertions are formulas of first order logic. Programming language variables can directly occur in the formula. The Hoare logic rules map the effect of a statement to the assertions via syntactic substitutions. For example, consider the assignment rule:

$$\{Q[e/x]\}\ x := e\ \ \{Q\}.$$

Often the intuitive meaning of a Hoare logic rule is revealed by reading it backwards. If after the assignment the assertion Q holds, then before the assignment $Q[e/x]$ holds, which we obtain out of Q by replacing all occurrences of variable x by e. For example, if the postcondition Q is $x < 5$ then the precondition is $e < 5$. The appeal of this approach is the direct syntactic correspondence between the variables in the assertion and the variables of the program. The assertion is only modified in those parts that mention the variable that is assigned to. The situation gets more involved as we introduce procedures with parameter passing and local variables. To properly reason about procedure calls [43, 7], fresh variable names have to be invented that neither occur in the body of the procedure nor in the assertions. Another problem is aliasing that occurs when dealing with arrays or heap. Two syntactically different pointers p and q can point to the same heap location. An assignment to the heap via p also modifies the content of q. Thus a simple syntactic substitution of all occurrences of p is not sufficient to describe the effect. The problem is the indirection introduced by dereferencing a pointer. Not the pointer is modified but the heap cell it points to. However, the heap is not explicitly visible in the programming language. As an example consider the assignment $*p := 4$, where $*p$ means dereferencing pointer p. If we want to derive $\{*p = 4 \wedge *q = 5\}$ as a postcondition it is not sufficient to just substitute $*p$ by 4 to get the precondition $\{4 = 4 \wedge *q = 5\}$. The assignment can also have an effect on $*q$ if p equals q. The effect of the assignment is not determined by the syntactic occurrence of $*p$, but by the actual *value* of p, the heap cell it points to. This dependency of the value can be introduced as a case distinction in the assertion logic. For every dereference of a pointer that is syntactically different from p we introduce such a case distinction. In our example we arrive at the precondition $\{4 = 4 \wedge \mathit{if}\ q = p\ \mathit{then}\ 4 = 5\ \mathit{else}\ *q = 5\}$. To satisfy this assertion p and q have to point to different heap cells in the beginning. The case distinction on the pointer value is the essence of dealing with aliasing. In the end all approaches boil down to it [20, 101, 93, 67, 13].

The first question to answer when formalising a Hoare logic is how to represent assertions. Traditionally assertions are first order formulas where the effect of the program is described by substitutions. Directly adapting this scheme to HOL means to formalise the assertion logic and the notion of substitution. To avoid this extra layer of formalisation we follow the *extensional* approach [76] and directly map the assertion logic to HOL [71, 111, 46, 84, 77, 67]. This immediately makes the existing infrastructure of the theorem prover available for the assertion logic. An assertion describes a set of states, hence we can directly use the set comprehension of HOL:

$$\{s.\ P\ s\}.$$

This approach is perfectly suited for the generic nature of Simpl. The state space is polymorphic and there is no formal notion of "program variable"on that level. Therefore a substitution based assertion logic that would introduce variables would destroy this genericity. Assignments are special *Basic* statements. The generalised assignment rule has the following form:

$$\{s.\ f\ s \in Q\}\ (Basic\ f)\ Q.$$

We avoid the set brackets around the postcondition, since Q already is a set in HOL. For every concrete postcondition the standard set comprehension notation automatically introduces the set brackets, like in the precondition. The states satisfying the precondition are exactly those states that lead to the postcondition by executing f. The precondition anticipates the semantic effect of f. This rule is sound for all *Basic* actions. It automatically works for assignments or pointer updates as they can all be encoded in f. However, this semantical rule poses the question if we lose the original benefit of Hoare logics to abstract from the semantics and directly work on the assertion syntax. At first this is true, but we can regain the benefits by exploiting the infrastructure of Isabelle. Let us consider the assignment $\mathsf{m} := e$ and the postcondition $\{s.\ m\ s < 4 \wedge n\ s = 5\}$.

Remember that the assignment is mapped to *Basic* $(\lambda s.\ s(\!|m := e|\!))$. Therefore instantiation of the Hoare Rule for *Basic* yields:

$$\{s.\ s(\!|m := e|\!) \in \{s.\ m\ s < 4 \wedge n\ s = 5\}\}\ \mathsf{m} := e\ \{s.\ m\ s < 4 \wedge n\ s = 5\}.$$

Now we can employ Isabelle's simplifier to derive a more appealing precondition. First, we can substitute the state update into the set, by rewriting with the equation $(a \in \{x.\ P\ x\}) = P\ a$:

$$\{s.\ m\ (s(\!|m := e|\!)) < 4 \wedge n\ (s(\!|m := e|\!)) = 5\}\ \mathsf{m} := e\ \{s.\ m\ s < 4 \wedge n\ s = 5\}.$$

Moreover, we can simplify the record selections of the state update, $m\ (s(\!|m := e|\!))$ becomes e and $n\ (s(\!|m := e|\!))$ rewrites to $n\ s$. We arrive at:

$$\{s.\ e < 4 \wedge n\ s = 5\}\ \mathsf{m} := e\ \{s.\ m\ s < 4 \wedge n\ s = 5\}.$$

This corresponds to the syntactic substitution that we expect from the original Hoare logic assignment rule. Variable m is substituted by e and n is left unchanged. We provide syntactic sugar to hide the abstraction over s and the selection of record fields from s. The special brackets $\{\!|\ldots|\!\}$ implicitly abstract over the state space, and the selection can be abbreviated with $\acute{m}$, as in the assignment. With our syntactic convention to use a sans-serif font instead of the prefixed symbol $\acute{}$, we can write $\{\!|\mathsf{m} = 4|\!\}$ for $\{s.\ m\ s = 4\}$. So finally we get back to the textbook version for the assignment:

$$\{\!|e < 4 \wedge \mathsf{n} = 5|\!\}\ \mathsf{m} := e\ \{\!|\mathsf{m} < 4 \wedge \mathsf{n} = 5|\!\}.$$

To summarise: We employ a shallow embedding of the assertions and directly represent them as HOL sets. This relieves us from the work to formalise an assertion logic and substitution operation, which can get quite involved when dealing with aliasing. The semantical version of the generalised assignment rule for *Basic* statements is straightforward and works out of the box for arbitrary state updates, including pointer manipulations. By instrumenting the infrastructure of Isabelle to simplify the assertions, we can still preserve the syntactic feeling of Hoare logic.

3.1 Partial Correctness

The Hoare logic for Simpl has to deal with abrupt termination. In the literature there are two different approaches. One [33, 53] splits the postcondition, the other [84] keeps the standard format of a Hoare triple. By holding a separate postcondition for normal and abrupt termination the Hoare logic keeps track of both possible control flows and the rules can directly manipulate either of them. By sticking to one postcondition reasoning about abrupt termination is forced into the assertion level. The postcondition itself has to be aware of abrupt and normal termination. It is a predicate on extended states $('s, 'f)$ *xstate* rather then on raw states $'s$. This makes assertions more complicated and additionally has the drawback, that the Hoare logic cannot look inside the postcondition, since we use a shallow embedding. A rule cannot take the postcondition apart into the portions about normal and abrupt termination but would instead add information about the current control flow into the assertions. For these reasons I decided to split the postcondition. The clear distinction between normal and abrupt termination results in a straightforward and clean calculus. Abrupt termination does not complicate reasoning about normal termination at all.

Definition 3.1 ► *Hoare logic for Simpl (partial correctness)*

The Hoare logic for Simpl is inductively defined by the rules in Figure 3.1. The judgement has the following form:

$$\Gamma,\Theta\vdash_{/F} P\ c\ Q,A,$$

where:

$\Gamma :: 'p \rightharpoonup ('s, 'p, 'f)\ com$	$P,Q,A :: 's\ set$
$\Theta :: ('s\ set \times 'p \times 's\ set \times 's\ set)\ set$	$c :: ('s, 'p, 'f)\ com$
$F :: 'f\ set$	

To stick to common practice I continue talking about Hoare triples, although the judgement has more than tree components. P is the precondition, c a program fragment, Q the postcondition for normal termination, A the postcondition for abrupt termination and Γ is the procedure environment as in the operational semantics. Θ is a set of assumptions that contains those procedure specifications that are taken as granted while verifying c. It is used to handle mutually recursive procedures. The F is a set of faults. The intended meaning of the judgement is partial correctness modulo faults in F. A guarded statement has the form *Guard* $f\ g\ c$, where f is a certain fault of type $'f$, g the guard condition and c a statement. If the fault f is in F then we assume that the gaurd does not fail, otherwise the Hoare logic has to ensure that the guard holds. The set F is introduced to the Hoare calculus to facilitate the integration of automatic program analysers into the verification process (cf. Section 5). If the automatic tool has already proven that some guards never fail, this information can be exploited by the Hoare logic. The overall result should of course be a triple where F is the empty set, which means that we do not rely on the assumption that some guards cannot fail *per se*. An empty context Θ or empty postcondition A for abrupt termination or an empty set F of faults can be omitted. The intended formal semantics of a Hoare triple is defined by the notion of validity, which is written as $\Gamma \models_{/F} P\ c\ Q,A$:

◂ Definition 3.2 *Validity (partial correctness)*

$$\begin{array}{l}\Gamma \models_{/F} P\ c\ Q,A \equiv \\ \quad \forall s\ t.\ \Gamma\vdash \langle c,s\rangle \Rightarrow t \longrightarrow s \in \mathit{Normal}\ \text{`}\ P \longrightarrow t \notin \mathit{Fault}\ \text{`}\ F \longrightarrow t \in \mathit{Normal}\ \text{`}\ Q \cup \mathit{Abrupt}\ \text{`}\ A\end{array}$$

Given an execution of statement c from initial state s to final state t, provided that s is a *Normal* state satisfying the precondition P, provided that t is non of the *Fault* states in F, then t either becomes a *Normal* state satisfying Q or an *Abrupt* state satisfying A. *Fault* or *Stuck* states are excluded by this semantics. If the set F is empty then the Hoare logic has to guarantee that no fault occurs. In a sense the calculus is not completely "partial". Validity does not take context Θ into account. The context is only needed in intermediate steps of the derivation. In the end the Hoare triple is proven relative to an empty context. Validity and the Hoare logic are related by two important theorems that are proven in Sections 3.1.3 and 3.1.4:

- *Soundness*: $\Gamma\vdash_{/F} P\ c\ Q,A \longrightarrow \Gamma\models_{/F} P\ c\ Q,A$
  We can only derive valid Hoare triples out of the empty context.
- *Completeness*: $\Gamma\models_{/F} P\ c\ Q,A \longrightarrow \Gamma\vdash_{/F} P\ c\ Q,A$
  We can derive every valid Hoare triple out of the empty context.

The Hoare logic rules are divided into two parts, for each syntactic construct of Simpl there is exactly one rule and there are two structural rules, the consequence rule and the assumption rule. The rules are written in a weakest precondition style, which means that the postcondition consists of plain variables Q and A whereas the precondition is obtained from the postcondition. Given a program and a postcondition the application of the rules yields the weakest precondition. The Rules SKIP, BASIC, SEQ, COND and WHILE are standard.

How to deal with procedures is elaborated in Section 3.1.2.

For a guarded command *Guard f g c* we can assume that the guard holds when we verify the body c. The standard GUARD Rule also requires to prove that the guard actually never fails. However, when the fault belongs to the set F the guard can always be regarded as guarantee and hence the GUARANTEE Rule treats it as an additional assumption.

The postcondition for abrupt termination A is left untouched and handed over to the sub-statements by most of the rules. Only the rules for *Throw* and *Catch* consider it. In case of a *Throw* it has to stem from the precondition. The THROW Rule is dual to the SKIP Rule, where the postcondition Q for normal termination comes from the precondition. Similarly, the CATCH Rule is dual to sequential composition. In case of *Catch*, the intermediate assertion R connects the precondition of the second statement with abrupt termination of the first statement. For *Seq* however, it is connected with normal termination of the first statement.

For the nondeterministic *Spec* we have to establish the postcondition for every possible successor state. To avoid getting stuck there must be at least one successor state.

$$\frac{}{\Gamma,\Theta\vdash_{/F} Q\ \mathit{Skip}\ Q,A}\ (\textsc{Skip}) \qquad \frac{}{\Gamma,\Theta\vdash_{/F} \{s.\ f\ s \in Q\}\ (\mathit{Basic}\ f)\ Q,A}\ (\textsc{Basic})$$

$$\frac{\Gamma,\Theta\vdash_{/F} P\ c_1\ R,A \qquad \Gamma,\Theta\vdash_{/F} R\ c_2\ Q,A}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{Seq}\ c_1\ c_2)\ Q,A}\ (\textsc{Seq})$$

$$\frac{\Gamma,\Theta\vdash_{/F} (P \cap b)\ c_1\ Q,A \qquad \Gamma,\Theta\vdash_{/F} (P \cap -b)\ c_2\ Q,A}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{Cond}\ b\ c_1\ c_2)\ Q,A}\ (\textsc{Cond}) \qquad \frac{\Gamma,\Theta\vdash_{/F} (P \cap b)\ c\ P,A}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{While}\ b\ c)\ (P \cap -b),A}\ (\textsc{While})$$

$$\frac{(P,\ p,\ Q,\ A) \in \mathit{Specs} \qquad \forall (P,\ p,\ Q,\ A){\in}\mathit{Specs}.\ p \in \mathit{dom}\ \Gamma \wedge \Gamma,\Theta \cup \mathit{Specs}{\vdash_{/F}}\ P\ (\mathit{the}\ (\Gamma\ p))\ Q,A}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{Call}\ p)\ Q,A}\ (\textsc{CallRec})$$

$$\frac{\Gamma,\Theta\vdash_{/F} (g \cap P)\ c\ Q,A}{\Gamma,\Theta\vdash_{/F} (g \cap P)\ (\mathit{Guard}\ f\ g\ c)\ Q,A}\ (\textsc{Guard}) \qquad \frac{f \in F \qquad \Gamma,\Theta\vdash_{/F} (g \cap P)\ c\ Q,A}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{Guard}\ f\ g\ c)\ Q,A}\ (\textsc{Guarantee})$$

$$\frac{}{\Gamma,\Theta\vdash_{/F} A\ \mathit{Throw}\ Q,A}\ (\textsc{Throw}) \qquad \frac{\Gamma,\Theta\vdash_{/F} P\ c_1\ Q,R \qquad \Gamma,\Theta\vdash_{/F} R\ c_2\ Q,A}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{Catch}\ c_1\ c_2)\ Q,A}\ (\textsc{Catch})$$

$$\frac{}{\Gamma,\Theta\vdash_{/F} \{s.\ (\forall t.\ (s,\ t) \in r \longrightarrow t \in Q) \wedge (\exists t.\ (s,\ t) \in r)\}\ (\mathit{Spec}\ r)\ Q,A}\ (\textsc{Spec})$$

$$\frac{\forall s{\in}P.\ \Gamma,\Theta\vdash_{/F} P\ (c_s\ s)\ Q,A}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{DynCom}\ c_s)\ Q,A}\ (\textsc{DynCom})$$

$$\frac{\forall s{\in}P.\ \exists P'\ Q'\ A'.\ \Gamma,\Theta\vdash_{/F} P'\ c\ Q',A' \wedge s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A}{\Gamma,\Theta\vdash_{/F} P\ c\ Q,A}\ (\textsc{Conseq}) \qquad \frac{(P,\ p,\ Q,\ A) \in \Theta}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{Call}\ p)\ Q,A}\ (\textsc{Asm})$$

Figure 3.1: Hoare logic for partial correctness

Since the dynamic command depends on the state, the Hoare triple has to hold for every possible instance stemming from a state in the precondition.

The next section is dedicated to a detailed discussion of the consequence rule, followed by some explanation how to deal with mutually recursive procedures. There the assumption rule comes in.

3.1.1 "The" Consequence Rule

A consequence rule allows to adapt the pre- and postcondition of a specification. A Hoare triple can thus be reused in different contexts without having to reprove the body. Reuse is a crucial ingredient to supply modular reasoning on the granularity of procedures. A procedure should only be specified and proven correct once and upon a call to this procedure its specification can be inserted into the current proof. The most common consequence rule allows to strengthen the precondition or weaken the postcondition:

$$\frac{\Gamma,\Theta\vdash_{/F} P'\ c\ Q',A' \qquad P \subseteq P' \qquad Q' \subseteq Q \qquad A' \subseteq A}{\Gamma,\Theta\vdash_{/F} P\ c\ Q,A}$$

Actually there are two ways to read a consequence rule which influences the intuitive view on *weakening* or *strengthening*, namely from the premises to the conclusion or vice versa. In the first case we assume that we have given the specification $\Gamma,\Theta\vdash_{/F} P'\ c\ Q',A'$ and want to adapt it. We are allowed to strengthen the precondition and weaken the postcondition. The reading from the conclusion to the premises is motivated by backwards reasoning. Consider a proof, where we have to show $\Gamma,\Theta\vdash_{/F} P\ c\ Q,A$. The (backwards) application of the consequence rule results in a new proof state where we have to show $\Gamma,\Theta\vdash_{/F} P'\ c\ Q',A'$ for a weaker precondition but a stronger postcondition. So the commonly used view to strengthen the precondition and weaken the postcondition refers to the first reading, from the premises to the conclusion.

Auxiliary variables are crucial for specifications in Hoare logics. Traditionally auxiliary variables are those variables in the assertions that do not appear in the program itself. They are merely used for specification. The pre- and postcondition of a Hoare triple are predicates on the initial and final state of the program. Using auxiliary variables allows to relate the pre- and the postcondition. Consider a simple state space with only one variable n and the following specification of the identity:

$$\Gamma\vdash \{\!|\mathsf{n} = n|\!\}\ c\ \{\!|\mathsf{n} = n|\!\}.$$

The value of variable n is preserved by the identity. Formally the program variable n is fixed to the auxiliary (or logical) variable n. The postcondition again refers to the auxiliary variable n to relate the output to the input. The problem with the auxiliary variable is that we cannot derive

$$\Gamma\vdash \{\!|\mathsf{n} = n - 1|\!\}\ c\ \{\!|\mathsf{n} = n - 1|\!\}$$

with the simple consequence rule above, since this would boil down to prove that $n = n - 1$. This situation naturally occurs when verifying recursive procedures. We may assume the specification of the procedure while verifying its body. Imagine n to be the input parameter of the factorial. We assume that the specification works for a value n, but for the recursive call we need the specification for $n - 1$, which we cannot obtain with the simple consequence rule. The calculus becomes incomplete. For this reason Kleymann [104, 60, 61] argues that auxiliary variables have to be taken into account by the Hoare logic and the assertions. He explicitly introduces the dependency of assertions on the auxiliary variables. Assertions are no longer just predicates on the state space but also on the auxiliary variables. He introduces a generalised consequence rule inspired by Morris [73] that allows to adapt the auxiliary variables and obtains a complete calculus. His approach was adapted by Oheimb [83, 84] and by Nipkow [78] where assertions are modelled as predicates of type $'a \Rightarrow 's \Rightarrow bool$, where $'a$ is the type of auxiliary variables and $'s$ the type of the state space. So formally there is only one auxiliary variable, which somehow has to contain all necessary auxiliary variables. To prove completeness of the Hoare logic, the type of the auxiliary variable is identified with the state space type. The auxiliary variable is used to fix the state of the precondition, to refer to the initial state in the postcondition. Unfortunately the fixed type of the auxiliary variable makes the calculus clumsy in practice. All specifications have to be forced in a format where only one auxiliary variable is used, which is a kind of shadow state. Later Oheimb and Nipkow [85] refine their rules and get rid of the explicit dependency of the assertions on the auxiliary variables. Assertions now formally only depend on the state space: $'s \Rightarrow bool$. Unfortunately the rule of consequence and the rule to handle recursive procedures again force a fixed type for the auxiliary variable. The dependency is only moved to the meta level of HOL. The situation is somehow annoying. The consequence rule is sound without fixing it to a specific type, but the formalisation in HOL forces it to a fixed type thus making the calculus hard to use. This awkward behaviour of the embedding of the Hoare calculus in HOL indicates that there is still an issue with the consequence rule and the rule for recursive procedures. It seems that auxiliary variables have been taken "too serious". Indeed the Hoare logic for Simpl does not introduce auxiliary variables at all. Nevertheless a consequence rule à la Kleymann can be derived from the core rules. Therefore the type of an auxiliary variable is not fixed and can be different for each use of the rule. Every specification can introduce any number of auxiliary variables of any type to express the desired property. In the following I motivate and introduce the Simpl consequence rule and compare it to other rules.

In the identity specification $\Gamma\vdash \{\!|\mathsf{n} = n|\!\}\ c\ \{\!|\mathsf{n} = n|\!\}$ the auxiliary variable n is a free variable of HOL. Thus it is logically equivalent to this universally quantified version:

$$\forall n.\ \Gamma\vdash \{\!|\mathsf{n} = n|\!\}\ c\ \{\!|\mathsf{n} = n|\!\}.$$

Our initial attempt to derive $\Gamma\vdash \{\!|\mathsf{n} = n - 1|\!\}\ c\ \{\!|\mathsf{n} = n - 1|\!\}$ now boils down to simple instantiation of the universally quantified variable. So this problem can already be solved on the level of HOL and there is no need for a special treatment inside the Hoare logic. However, can we derive $\Gamma\vdash \{\!|\mathsf{n} = m \wedge 0 \le \mathsf{n}|\!\}\ c\ \{\!|0 \le \mathsf{n}|\!\}$ from this specification? We suppose to be in an initial state were variable n has a positive value m. Can we transfer the information that n is positive to the postcondition? We first instantiate the universally quantified n in our specification to m. Then we apply the consequence rule and have to show $\{\!|\mathsf{n} = m \wedge 0 \le \mathsf{n}|\!\} \subseteq \{\!|\mathsf{n} = m|\!\}$ for the precondition, and $\{\!|\mathsf{n} = m|\!\} \subseteq \{\!|0 \le \mathsf{n}|\!\}$ for the postcondition. The case for the precondition is trivial, but for the postcondition we are stuck. We only know that $\mathsf{n}=m$ and have no means to derive that m is positive. The problem is that the pre- and postconditions are separated by the consequence rule. From the postcondition of the specification we know that the value of n is still m. However, we cannot make use of the general knowledge of the new precondition about m being positive while solving the constraint for the postcondition.

We can try to reformulate the target Hoare triple $\Gamma\vdash \{\!|\mathsf{n} = m \wedge 0 \le \mathsf{n}|\!\}\ c\ \{\!|0 \le \mathsf{n}|\!\}$. Since m is bound outside of the whole Hoare triple, we can reuse it in the postcondition. We reformulate the postcondition $\{\!|0 \le \mathsf{n}|\!\}$ to $\{\!|0 \le m \longrightarrow 0 \le \mathsf{n}|\!\}$. Since m is a state independent logical variable and we can derive from the precondition that m is positive, we can put this assumption as a hypothesis to the postcondition. Now we can indeed derive $\Gamma\vdash \{\!|\mathsf{n} = m \wedge 0 \le \mathsf{n}|\!\}\ c\ \{\!|0 \le m \longrightarrow 0 \le \mathsf{n}|\!\}$ from our identity specification. Again we instantiate n with m and apply the consequence rule. The constraint on the precondition stays the same, but know for the postcondition we get $\{\!|\mathsf{n} = m|\!\} \subseteq \{\!|0 \le m \longrightarrow 0 \le \mathsf{n}|\!\}$. We can assume both $\mathsf{n}=m$ and $0 \le m$ to conclude that $0 \le \mathsf{n}$, which is trivial.

If we abstract from the predicate $0 \le \mathsf{n}$ and the postcondition we arrive at two Hoare triples describing the same property:

- $\Gamma\vdash \{\!|\mathsf{n} = m \wedge P\ \mathsf{n}|\!\}\ c\ \{\!|Q\ \mathsf{n}|\!\}$
- $\Gamma\vdash \{\!|\mathsf{n} = m \wedge P\ \mathsf{n}|\!\}\ c\ \{\!|P\ m \longrightarrow Q\ \mathsf{n}|\!\}$

Note that in the first Hoare triple the auxiliary (logical) variable m only appears in the precondition. It was introduced to give the initial value of n a name so that we can instantiate the identity specification. Conceptually we mean $\Gamma\vdash \{\!|P\ \mathsf{n}|\!\}\ c\ \{\!|Q\ \mathsf{n}|\!\}$. In the second Hoare triple the logical variable m is used to connect the pre- and the postcondition. Like in the identity specification it can be interpreted as universally quantified outside of the Hoare triple. With this slightly more general view, we ultimately compare the following two schemes of specification:

- $\Gamma\vdash P\ c\ Q$
- $\forall Z.\ \Gamma\vdash \{s.\ s = Z \wedge s \in P\}\ c\ \{t.\ Z \in P \longrightarrow t \in Q\}$ (∗)

The second one is a more elaborate version, that explicitly fixes the pre-state to the auxiliary variable Z and includes the knowledge about this pre-state in the postcondition. Semantically both specifications are equivalent:

$$\Gamma\models_{/F} P\ c\ Q{,}A = (\forall Z.\ \Gamma\models_{/F} \{s.\ s = Z \wedge s \in P\}\ c\ \{t.\ Z \in P \longrightarrow t \in Q\}{,}\{t.\ Z \in P \longrightarrow t \in A\})$$ ◂ Lemma 3.1

The simple consequence rule allows to derive the second specification from the first, since $\{s.\ s = Z \wedge s \in P\} \subseteq P$ and $Q \subseteq \{t.\ Z \in P \longrightarrow t \in Q\}$. Unfortunately we are not able to derive the first specification from the second one. We already fail to find a proper instance of Z since the initial state is only "inside" the first precondition P. The simple consequence rule gives us no means to instantiate Z with a $s \in P$. Even if the first specification fixes the pre-state to a logical variable and is given in the right format: $\Gamma\vdash \{s.\ s = Z \wedge s \in P\}\ c\ Q$, we cannot deduce the constraint for the postcondition: $\{t.\ Z \in P \longrightarrow t \in Q\} \subseteq Q$. We can instantiate the second specification with Z, but as in the identity example we have no means to discharge $Z \in P$, since the simple consequence rule strictly separates the pre- and the postcondition. Not all semantically equivalent specifications can be derived from each other using the simple consequence rule. Sticking to the nomenclature of Kleymann the calculus is not *adaptation complete*. Note that this does not necessarily imply that the calculus is incomplete. It may still be possible to derive the desired property without the given specification, by reproving the body.

A first step to remedy this situation is by a more liberal side-condition in the consequence rule:

$$\frac{\Gamma{,}\Theta\vdash_{/F} P'\ c\ Q'{,}A' \qquad \forall s{\in}P.\ s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A}{\Gamma{,}\Theta\vdash_{/F} P\ c\ Q{,}A}$$

We only have to establish weakening of the postcondition under the additional assumption of an initial state satisfying the new precondition P. Therefore we can derive $\Gamma\vdash \{s.\ s = Z \wedge s \in P\}\ c\ Q$ from specification (∗). The critical postcondition strengthening is easy under the assumption of the new precondition:

$$\forall s{\in}\{s.\ s = Z \wedge s \in P\}.\ \{t.\ Z \in P \longrightarrow t \in Q\} \subseteq Q$$

$Z \in P$ can be established, since we know both $s \in P$ and $s = Z$ from the precondition. Note that s is fixed to Z, which is a free variable or bound outside of the new triple. That is why the postconditions in the rule above formally do not have to depend on state s.

However, we are still not able to derive $\Gamma\vdash P\ c\ Q$. Basically we need to instantiate Z with a $s \in P$, but the new consequence rule still strictly separates the side-condition from the Hoare triple in the premise. The canonical solution is to integrate the specification triple into the side-condition, which leads to the Simpl consequence rule:

$$\boxed{\frac{\forall s{\in}P.\ \exists P'\,Q'\,A'.\ \Gamma{,}\Theta{\vdash_{/F}}\,P'\ c\ Q'{,}A' \wedge s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A}{\Gamma{,}\Theta{\vdash_{/F}}\,P\ c\ Q{,}A}}$$

We can select a suitable specification under the assumption of a $s \in P$. In the example we can instantiate Z of the specification with s and arrive at the following trivial side-condition:

$$\forall s{\in}P.\ s \in \{s.\ s = s \wedge s \in P\} \wedge \{t.\ s \in P \longrightarrow t \in Q\} \subseteq Q$$

Oheimb [84] already mentions a similar consequence rule, where the specification triple can be selected under the assumption of the new precondition. However, as the auxiliary variable is explicit in his assertions the side-condition also deals with them, which leads to a fixed type for the auxiliary variable.

The general consequence rule gives complete freedom to select the specification depending on the new precondition. The consequence rule of Kleymann can be obtained from it by restricting this freedom to the adaptation of the auxiliary variable. It again disentangles the specification triple from the side-condition, but still links them together via the auxiliary variable:

$$\frac{\begin{array}{c}\forall Z.\ \Gamma{,}\Theta{\vdash_{/F}}\,(P'\,Z)\ c\ (Q'\,Z){,}(A'\,Z)\\ \forall s{\in}P.\ \exists Z.\ s \in P'\,Z \wedge Q'\,Z \subseteq Q \wedge A'\,Z \subseteq A\end{array}}{\Gamma{,}\Theta{\vdash_{/F}}\,P\ c\ Q{,}A}\ (\textsc{ConseqAux})$$

The auxiliary variable Z is universally quantified in the specification. The side-condition allows to select a suitable Z from the initial state. Since this consequence rule is a derived rule and not part of the inductive definition of the core calculus, HOL puts no restrictions on the type of Z. The auxiliary variable only appears in the premises. Therefore its type is not constrained by the conclusion at all. If we directly include this rule to the inductive definition we would have to fix the type of Z or somehow make it visible in the conclusion. In order to preserve soundness, the right-hand side of a definition may not introduce new type variables. Otherwise the value of the constant would depend on a hidden type that is not visible from the constant itself.

All the consequence rules seen so far work for both partial and total correctness. All of them ensure that the new precondition implies the precondition of the specification. As soon as the specification is proven, termination is guaranteed. Kleymann [61] also introduces a consequence rule that only works for partial correctness. It allows to bypass the specification if the new postcondition can directly be established. Ignoring abrupt termination this consequence rule reads as follows:

$$\frac{\forall Z.\ \Gamma{,}\Theta{\vdash_{/F}}\,(P'\,Z)\ c\ (Q'\,Z) \qquad \forall s{\in}P.\ \forall t.\ (\forall Z.\ s \in P'\,Z \longrightarrow t \in Q'\,Z) \longrightarrow t \in Q}{\Gamma{,}\Theta{\vdash_{/F}}\,P\ c\ Q{,}A}$$

In order to prove that the final state t satisfies the new postcondition Q we may or may not use the specification. If we want to use it we have to find a proper instance of Z and prove $s \in P'\,Z$ to get hold of $t \in Q'\,Z$. If we can directly provide the new postcondition, like the trivial one $\{\!|True|\!\}$, we do not have to take this detour and can immediately satisfy the side-condition. This consequence rule allows to circumvent to establish the precondition of the specification. Therefore total correctness is no longer guaranteed. In our setting such a partial correctness rule does not work. The semantics of our Hoare logic is more total in the sense that even partial correctness ensures that we do not end in a *Fault* or *Stuck* state. However, the following rule works for partial correctness:

$$\frac{\begin{array}{l}\forall s \in P.\ \forall t.\ \exists P'\,Q'\,A'.\ \Gamma{,}\Theta\vdash_{/F} P'\ c\ Q'{,}A' \wedge \\ \qquad ((s \in P' \longrightarrow t \in \mathit{Normal}\ `\ Q' \cup \mathit{Abrupt}\ `\ A') \longrightarrow t \in \mathit{Normal}\ `\ Q \cup \mathit{Abrupt}\ `\ A)\end{array}}{\Gamma{,}\Theta\vdash_{/F} P\ c\ Q{,}A}$$

This rule mimics Kleymanns partial correctness rule, but additionally ensures that we do not end up in a *Fault* or *Stuck* state. Again we can avoid to establish the precondition of the specification if we can directly show the new postcondition. This consequence rule is strictly more liberal then the general consequence rule. We can derive the general consequence rule from it but not vice versa. However, this rule is a kind of monster. It breaks the abstraction level of our Hoare logic, since it introduces direct reasoning on the extended states. Moreover, it is unclear how we could practically make use of the possibility to circumvent the precondition of the specification. How can we argue that the final state is neither *Fault* nor *Stuck* without using the postcondition of the specification? These are the reasons for me to avoid putting this consequence rule to the core calculus.

3.1.2 Recursive Procedures

The basic idea to handle recursive procedures, is to assume the specification while verifying the procedure body. Technically a procedure specification can be assumed, by augmenting the set Θ of assumptions with it. Every recursive call can then be directly handled by the assumption rule Asm. To handle mutually recursive procedures we simultaneously add all the specifications to the assumptions and prove their bodies correct. This is the nature of the CallRec Rule. Any number of procedure specifications *Specs* can be added to the assumptions Θ, as long as our target specification is among them and we prove that all procedure bodies meet their specification. Moreover, to avoid getting stuck all procedures must be defined in context Γ.

The general rule can be specialised to one procedure by instantiating *Specs* with the single specification $\{(P, p, Q, A)\}$. Then it simplifies to the familiar rule:

$$\frac{p \in \mathit{dom}\ \Gamma \qquad \Gamma{,}\Theta \cup \{(P, p, Q, A)\}\vdash_{/F} P\ (\mathit{the}\ (\Gamma\ p))\ Q{,}A}{\Gamma{,}\Theta\vdash_{/F} P\ (\mathit{Call}\ p)\ Q{,}A}$$

As already indicated in the discussion about the consequence rule this version is often to restrictive, since it only moves the exact specification to the assumptions. As soon as the specification uses auxiliary variables a recursive call might require different instances of the auxiliary variables. Take the factorial as an example:

$$\Gamma\vdash \{\!|\mathsf{n} = n|\!\}\ \mathit{Call}\ \text{"}\mathit{fac}\text{"}\ \{\!|\mathsf{n} = n!|\!\}.$$

If we only add $(\{\!|\mathsf{n} = n|\!\},\text{"}\mathit{fac}\text{"},\{\!|\mathsf{n} = n!|\!\},\{\})$ to the assumptions we cannot handle the recursive call which requires $(\{\!|\mathsf{n} = n - 1|\!\},\text{"}\mathit{fac}\text{"},\{\!|\mathsf{n} = n - 1!|\!\},\{\})$. However, the general rule for recursion does not forbid adding more than one instance of the specification to the assumptions. We can add all the instances at once:

$$\mathit{Specs} = \bigcup_n\ \{(\{\!|\mathsf{n} = n|\!\},\text{"}\mathit{fac}\text{"},\{\!|\mathsf{n} = n!|\!\},\{\})\}.$$

Now we can handle every possible recursive call to the factorial. Generalising this idea we can derive this version to handle one recursive procedure with auxiliary variables:

$$\frac{p \in \mathit{dom}\ \Gamma \qquad \forall Z.\ \Gamma,\Theta \cup (\bigcup_Z \{(P\ Z, p, Q\ Z, A\ Z)\}) \vdash_{/F} (P\ Z)\ (\mathit{the}\ (\Gamma\ p))\ (Q\ Z),(A\ Z)}{\Gamma,\Theta \vdash_{/F} (P\ Z)\ (\mathit{Call}\ p)\ (Q\ Z),(A\ Z)}$$

To handle mutually recursive procedures we can derive a variant that handles all procedures and auxiliary variables at once:

$$\frac{\begin{array}{c}\mathcal{P} \subseteq \mathit{dom}\ \Gamma \\ \forall p{\in}\mathcal{P}.\ \forall Z.\ \Gamma,\Theta \cup (\bigcup_{p\in\mathcal{P}} \bigcup_Z \{(P\ p\ Z, p, Q\ p\ Z, A\ p\ Z)\}) \vdash_{/F} (P\ p\ Z)\ (\mathit{the}\ (\Gamma\ p))\ (Q\ p\ Z),(A\ p\ Z)\end{array}}{\forall p{\in}\mathcal{P}.\ \forall Z.\ \Gamma,\Theta \vdash_{/F} (P\ p\ Z)\ (\mathit{Call}\ p)\ (Q\ p\ Z),(A\ p\ Z)}$$

The assertions P, Q and A are indexed by the procedure name and depend on the auxiliary variable Z. $\mathcal{P}$ is the set of mutually recursive procedures. The general CallRec Rule allows to derive this rule, but itself appears to be much simpler. Moreover, it does not introduce the complication that we have to be explicit about the type of Z as discussed for the consequence rule.

3.1.3 Soundness

Soundness of the Hoare logic means that we can only derive semantically valid triples within the calculus:

$$\Gamma \vdash_{/F} P\ c\ Q,A \longrightarrow \Gamma \models_{/F} P\ c\ Q,A.$$

We assume to have a derivation of a Hoare triple and want to show that it is indeed valid according to definition 3.2. We follow the proof of Oheimb [84] by induction on the Hoare logic derivation. Expanding the definition of validity we have given an execution from an initial state satisfying the precondition to a final state that does not belong to the set of excluded faults and have to show that the final state indeed satisfies the postcondition and neither becomes *Fault* nor *Stuck*. Since the Hoare logic rules for atomic statements like *Basic* mimic the semantics anyway these cases are straightforward. For compositional statements the Hoare Logic exactly follows the syntax and thus we can assume validity for the sub-statements and argue on their composition. The operational semantics itself follows the syntax, too, except for the loop and the procedure call. In case of the loop we can only assume validity of the loop body. To extend it to the iterated execution of the body we do a nested induction, but this time on the operational semantics. This works fine for the while loop, because of the regular pattern of its execution: a sequence of loop bodies until the condition becomes false. In case of the (mutually) recursive procedure calls there is no such regular pattern in the execution. Moreover, the Hoare logic exploits the assumptions Θ in the CallRec Rule, whereas validity does not involve them at all. We introduce an extended notion of validity that also takes them into account:

Definition 3.3 ► *Validity within context*

$$\Gamma,\Theta \models_{/F} P\ c\ Q,A \equiv (\forall (P, p, Q, A){\in}\Theta.\ \Gamma \models_{/F} P\ (\mathit{Call}\ p)\ Q,A) \longrightarrow \Gamma \models_{/F} P\ c\ Q,A$$

Provided that the specifications in Θ are valid, then the Hoare triple is also valid.

The basic idea to handle recursion is to argue on the recursion depth to justify that it is sound to assume correctness of the procedure specification while verifying the body. Since we deal with partial correctness we know that the program terminates, and thus the depth of recursion is finite and we can build an inductive argument on it. Unfortunately our operational semantics is not fine grained enough to talk about the depth or recursion. We introduce an auxiliary semantics that takes the depth of nested procedure calls into account and build the soundness proof on a refined notion of validity on the basis of this semantics.

◂ Definition 3.4
Big-step semantics with limit on nested procedure calls

The operational big-step semantics: $\Gamma\vdash \langle c,s\rangle \overset{n}{\Rightarrow} t$, *with natural number* n *as limit on nested procedure calls is defined inductively by the rules in Figure 3.2, where:*

$$\begin{aligned}\Gamma &:: 'p \rightharpoonup ('s, 'p, 'f)\ com\\ s,t &:: ('s, 'f)\ xstate\\ c &:: ('s, 'p, 'f)\ com\end{aligned}$$

$$\frac{}{\Gamma\vdash \langle Skip,Normal\ s\rangle \overset{n}{\Rightarrow} Normal\ s}\ (\textsc{Skip}) \qquad \frac{}{\Gamma\vdash \langle Basic\ f,Normal\ s\rangle \overset{n}{\Rightarrow} Normal\ (f\ s)}\ (\textsc{Basic})$$

$$\frac{\Gamma\vdash \langle c_1,Normal\ s\rangle \overset{n}{\Rightarrow} s' \qquad \Gamma\vdash \langle c_2,s'\rangle \overset{n}{\Rightarrow} t}{\Gamma\vdash \langle Seq\ c_1\ c_2,Normal\ s\rangle \overset{n}{\Rightarrow} t}\ (\textsc{Seq})$$

$$\frac{s \in b \qquad \Gamma\vdash \langle c_1,Normal\ s\rangle \overset{n}{\Rightarrow} t}{\Gamma\vdash \langle Cond\ b\ c_1\ c_2,Normal\ s\rangle \overset{n}{\Rightarrow} t}\ (\textsc{CondTrue}) \qquad \frac{s \notin b \qquad \Gamma\vdash \langle c_2,Normal\ s\rangle \overset{n}{\Rightarrow} t}{\Gamma\vdash \langle Cond\ b\ c_1\ c_2,Normal\ s\rangle \overset{n}{\Rightarrow} t}\ (\textsc{CondFalse})$$

$$\frac{s \in b \qquad \Gamma\vdash \langle c,Normal\ s\rangle \overset{n}{\Rightarrow} s' \qquad \Gamma\vdash \langle While\ b\ c,s'\rangle \overset{n}{\Rightarrow} t}{\Gamma\vdash \langle While\ b\ c,Normal\ s\rangle \overset{n}{\Rightarrow} t}\ (\textsc{WhileTrue})$$

$$\frac{s \notin b}{\Gamma\vdash \langle While\ b\ c,Normal\ s\rangle \overset{n}{\Rightarrow} Normal\ s}\ (\textsc{WhileFalse})$$

$$\frac{\Gamma\ p = \lfloor bdy\rfloor \qquad \Gamma\vdash \langle bdy,Normal\ s\rangle \overset{n}{\Rightarrow} t}{\Gamma\vdash \langle Call\ p,Normal\ s\rangle \overset{n+1}{\Rightarrow} t}\ (\textsc{Call}) \qquad \frac{\Gamma\ p = None}{\Gamma\vdash \langle Call\ p,Normal\ s\rangle \overset{n+1}{\Rightarrow} Stuck}\ (\textsc{CallUndefined})$$

$$\frac{s \in g \qquad \Gamma\vdash \langle c,Normal\ s\rangle \overset{n}{\Rightarrow} t}{\Gamma\vdash \langle Guard\ f\ g\ c,Normal\ s\rangle \overset{n}{\Rightarrow} t}\ (\textsc{Guard}) \qquad \frac{s \notin g}{\Gamma\vdash \langle Guard\ f\ g\ c,Normal\ s\rangle \overset{n}{\Rightarrow} Fault\ f}\ (\textsc{GuardFault})$$

$$\frac{}{\Gamma\vdash \langle Throw,Normal\ s\rangle \overset{n}{\Rightarrow} Abrupt\ s}\ (\textsc{Throw})$$

$$\frac{\Gamma\vdash \langle c_1,Normal\ s\rangle \overset{n}{\Rightarrow} Abrupt\ s' \qquad \Gamma\vdash \langle c_2,Normal\ s'\rangle \overset{n}{\Rightarrow} t}{\Gamma\vdash \langle Catch\ c_1\ c_2,Normal\ s\rangle \overset{n}{\Rightarrow} t}\ (\textsc{Catch}) \qquad \frac{\Gamma\vdash \langle c_1,Normal\ s\rangle \overset{n}{\Rightarrow} t \qquad \neg\ isAbr\ t}{\Gamma\vdash \langle Catch\ c_1\ c_2,Normal\ s\rangle \overset{n}{\Rightarrow} t}\ (\textsc{CatchMiss})$$

$$\frac{(s,\ t) \in r}{\Gamma\vdash \langle Spec\ r,Normal\ s\rangle \overset{n}{\Rightarrow} Normal\ t}\ (\textsc{Spec}) \qquad \frac{\forall t.\ (s,\ t) \notin r}{\Gamma\vdash \langle Spec\ r,Normal\ s\rangle \overset{n}{\Rightarrow} Stuck}\ (\textsc{SpecStuck})$$

$$\frac{\Gamma\vdash \langle c_s\ s,Normal\ s\rangle \overset{n}{\Rightarrow} t}{\Gamma\vdash \langle DynCom\ c_s,Normal\ s\rangle \overset{n}{\Rightarrow} t}\ (\textsc{DynCom})$$

$$\frac{}{\Gamma\vdash \langle c,Fault\ f\rangle \overset{n}{\Rightarrow} Fault\ f}\ (\textsc{FaultProp}) \qquad \frac{}{\Gamma\vdash \langle c,Stuck\rangle \overset{n}{\Rightarrow} Stuck}\ (\textsc{StuckProp}) \qquad \frac{}{\Gamma\vdash \langle c,Abrupt\ s\rangle \overset{n}{\Rightarrow} Abrupt\ s}\ (\textsc{AbruptProp})$$

Figure 3.2: Big-step execution with limit on nested procedure calls

The rules are structurally equivalent to the normal big-step semantics. The limit n is just passed around, except for the procedure call, where it is decremented. A basic property of this semantics is monotonicity with respect to the limit for nested procedure calls.

Lemma 3.2 ► *Monotonicity*

If $\Gamma\vdash \langle c,s\rangle \overset{n}{\Rightarrow} t$ *and* $n \le m$ *then* $\Gamma\vdash \langle c,s\rangle \overset{m}{\Rightarrow} t$.

Proof. By induction on the execution up to depth n. □

Using this lemma we can show a kind of equivalence between both operational semantices:

Theorem 3.3 ► *Equivalence*

$$\Gamma\vdash \langle c,s\rangle \Rightarrow t = (\exists n.\ \Gamma\vdash \langle c,s\rangle \overset{n}{\Rightarrow} t)$$

Proof. Both directions are proven separately by induction on the respective semantics. □

By inserting the new semantics into our validity notions, we arrive at parameterised versions, that also take the recursion depth into account.

Definition 3.5 ► *Validity with limit*

$$\Gamma \overset{n}{\models}_{/F}\ P\ c\ Q,A \equiv$$
$$\forall s\ t.\ \Gamma\vdash \langle c,s\rangle \overset{n}{\Rightarrow} t \longrightarrow s \in \mathit{Normal}\ `\ P \longrightarrow t \notin \mathit{Fault}\ `\ F \longrightarrow t \in \mathit{Normal}\ `\ Q \cup \mathit{Abrupt}\ `\ A$$

And analogous for validity with respect to context Θ.

Definition 3.6 ► *Validity with limit and context*

$$\Gamma,\Theta \overset{n}{\models}_{/F}\ P\ c\ Q,A \equiv (\forall (P, p, Q, A){\in}\Theta.\ \Gamma \overset{n}{\models}_{/F}\ P\ (\mathit{Call}\ p)\ Q,A) \longrightarrow \Gamma \overset{n}{\models}_{/F}\ P\ c\ Q,A$$

Note that a Hoare triple is trivially valid in the sense of $\Gamma \overset{n}{\models}_{/F}\ P\ c\ Q,A$ if the program needs more than n recursive calls in order to terminate, because then there is no state s and t such that $\Gamma\vdash \langle c,s\rangle \overset{n}{\Rightarrow} t$. Using lemma 3.3 we arrive at the following relationship between the different notions of validity:

Lemma 3.4 ►

$$\Gamma\models_{/F}\ P\ c\ Q,A = (\forall n.\ \Gamma \overset{n}{\models}_{/F}\ P\ c\ Q,A)$$

A Hoare triple is valid in the usual sense if it is valid for all recursion depths. Therefore we only get one direction if we also take the context Θ into account:

Lemma 3.5 ►

$$(\forall n.\ \Gamma,\Theta \overset{n}{\models}_{/F}\ P\ c\ Q,A) \longrightarrow \Gamma,\Theta\models_{/F}\ P\ c\ Q,A$$

The other direction fails since we only get validity up to a fixed n for a specification in the assumptions of $\Gamma,\Theta \overset{n}{\models}_{/F}\ P\ c\ Q,A$, whereas we would need it for all n in order to apply Lemma 3.4. However, we are perfectly fine with this direction. By induction on a derivation $\Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A$ we prove validity for every recursion depth $\forall n.\ \Gamma,\Theta \overset{n}{\models}_{/F}\ P\ c\ Q,A$ and thus can conclude $\Gamma,\Theta\models_{/F}\ P\ c\ Q,A$ by Lemma 3.5.

Lemma 3.6 ► *Soundness with limit and context*

$$\Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A \longrightarrow (\forall n.\ \Gamma,\Theta \overset{n}{\models}_{/F}\ P\ c\ Q,A)$$

Proof. The proof is by induction on the Hoare logic derivation. I only describe the interesting cases, namely for the loop and the procedure call. The other cases are straightforward.

Case WHILE: As induction hypothesis we can assume validity of the loop body:

$$\forall n.\ \Gamma,\Theta \overset{n}{\models}_{/F}\ (P \cap b)\ c\ P,A.$$

We have to show $\Gamma,\Theta \models^{n}_{/F} P$ (*While b c*) $(P \cap - b),A$. According to Definition 3.6 of validity we have to consider an execution of the loop: $\Gamma \vdash \langle \textit{While b c},\textit{Normal s}\rangle \overset{n}{\Rightarrow} t$, and have to show $t \in \textit{Normal}\ `\ (P \cap - b) \cup \textit{Abrupt}\ `\ A$ under the assumptions of:

- a valid context: $\forall (P, p, Q, A){\in}\Theta.\ \Gamma \models^{n}_{/F} P\ (\textit{Call}\ p)\ Q,A$,
- the invariant for s: $s \in P$, and
- $t \notin \textit{Fault}\ `\ F$.

In case $s \notin b$ the loop is immediately exited and thus $t = \textit{Normal}\ s$ and we are finished since we know that the invariant holds for s. In case $s \in b$ first the loop body c is executed, followed by the recursive execution of *While b c*. From the induction hypothesis we know that the invariant P holds after execution of the loop body. To continue with the recursive loop we start a nested induction on the execution of the loop: Assuming $\Gamma \vdash \langle \textit{While b c},\textit{Normal s}\rangle \overset{n}{\Rightarrow} t$ and $s \in P$ and $t \notin \textit{Fault}\ `\ F$ we show that $t \in \textit{Normal}\ `\ (P \cap - b) \cup \textit{Abrupt}\ `\ A$. From the induction on the operational semantics we obtain an intermediate state r, where $\Gamma \vdash \langle c,\textit{Normal s}\rangle \overset{n}{\Rightarrow} r$ as well as $\Gamma \vdash \langle \textit{While b c},r\rangle \overset{n}{\Rightarrow} t$. From the outer hypothesis we get $r \in \textit{Normal}\ `\ P \cup \textit{Abrupt}\ `\ A$. In case r is an *Abrupt* state the execution and the proof is finished since $t = r$. In case r is *Normal* we know that the invariant holds and hence $r \in \textit{Normal}\ `\ P$. Thus we can apply the nested induction hypothesis to show the thesis.

Case CALLREC: We have to show $\Gamma,\Theta \models^{n}_{/F} P\ (\textit{Call}\ p)\ Q,A$ under the hypothesis:

- $(P, p, Q, A) \in \textit{Specs}$, and (∗)
- $\forall (P, p, Q, A){\in}\textit{Specs}.\ p \in \textit{dom}\ \Gamma \wedge (\forall n.\ \Gamma,\Theta \cup \textit{Specs} \models^{n}_{/F} P\ (\textit{the}\ (\Gamma\ p))\ Q,A)$. (∗∗)

To get hold of validity of the procedure body in hypothesis (∗∗) we have to discharge the context $\Theta \cup \textit{Specs}$. We unfold the definition of validity within context Θ and generalise the goal to all specifications in *Specs*:

$$(\forall (P, p, Q, A){\in}\Theta.\ \Gamma \models^{n}_{/F} P\ (\textit{Call}\ p)\ Q,A) \longrightarrow (\forall (P, p, Q, A){\in}\textit{Specs}.\ \Gamma \models^{n}_{/F} P\ (\textit{Call}\ p)\ Q,A).$$

Since we know from (∗) that the current procedure call is among *Specs* we are finished when this lemma is proven. We prove it by induction on the recursion depth n.

Case $n = 0$ is trivial, since there is no execution of a procedure call with a recursion limit of 0.

In case $n = m + 1$ we know $\forall (P, p, Q, A){\in}\Theta.\ \Gamma \models^{m+1}_{/F} P\ (\textit{Call}\ p)\ Q,A$. By the monotonicity Lemma 3.2 we get validity of the context Θ up to recursion depth m as well. Hence with the inner induction hypothesis we get validity up to recursion depth m for all procedure calls in *Specs*: $\forall (P, p, Q, A){\in}\textit{Specs}.\ \Gamma \models^{m}_{/F} P\ (\textit{Call}\ p)\ Q,A$. Putting the validity of the procedure calls in Θ and *Specs* together, we can discharge the context $\Theta \cup \textit{Specs}$ of the outer hypothesis (∗∗) and conclude that the procedure bodies are valid up to recursion depth m:

$$\forall (P, p, Q, A){\in}\textit{Specs}.\ \Gamma \models^{m}_{/F} P\ (\textit{the}\ (\Gamma\ p))\ Q,A.$$

Validity of the procedure bodies up to depth m corresponds to validity of the corresponding procedure calls up to depth $m + 1$:

$$\forall (P, p, Q, A){\in}\textit{Specs}.\ \Gamma \models^{m+1}_{/F} P\ (\textit{Call}\ p)\ Q,A.$$

□

Putting Lemmas 3.6 and 3.5 together we get:

Lemma 3.7 ► *Soundness within context*

$$\Gamma,\Theta \vdash_{/F} P\ c\ Q,A \longrightarrow \Gamma,\Theta \models_{/F} P\ c\ Q,A$$

Instantiating Θ with the empty context we arrive at the plain soundness theorem for the Hoare logic:

Theorem 3.8 ► *Soundness*

$$\Gamma \vdash_{/F} P\ c\ Q,A \longrightarrow \Gamma \models_{/F} P\ c\ Q,A$$

3.1.4 Completeness

Completeness of the Hoare logic is the converse question to soundness. Is every semantically valid triple derivable in the Hoare logic? Or formally:

$$\Gamma \models_{/F} P\ c\ Q,A \longrightarrow \Gamma \vdash_{/F} P\ c\ Q,A.$$

According to Cook [25] the more accurate term is *relative* completeness. Relative to the completeness of the underlying deductive system, in our case HOL. Consider a triple $\Gamma \models_{/F} \{\!|True|\!\}\ c\ \{\!|False|\!\},\{\!|False|\!\}$. This triple is valid if the program c does not terminate for any input. However, imagine program c is an universal program in the sense of recursion theory that has an undecidable halting problem. Proving the triple would mean to solve the halting problem within the underlying deductive system HOL. In this sense relative completeness only expresses that the Hoare logic rules do not introduce an additional source of incompleteness. A (relative) complete Hoare logic is the "right set" of rules, which allow to decompose a property about a program to a mere logical proposition in the assertion logic.

A related question is the *expressiveness* of the assertion logic, i.e. whether it is possible to express certain intermediate assertions and invariants that occur during a proof within the assertion logic. By our extensional approach with a shallow embedding of the assertions we have already solved this problem. In the completeness proof we can directly refer to the operational semantics in the assertions, without having to encode it in a special assertion language.

As intermediate step we prove that the *most general triple* (MGT) or *most general formula* [41] can be derived within the Hoare logic. The most general triple is a Hoare triple where the precondition does not exclude any states, and the postcondition is the set of final states that can be reached according to the operational semantics. The triple describes the same input/output relation as the operational semantics. All valid triples can be derived from the most general triple via the consequence rule. In our setting we have to restrict the set of initial states since our notion of partial correctness is not completely "partial". According to Definition 3.2 of validity only those executions are relevant which do not lead to a *Stuck* or *Fault* state that is not in the set F. The Hoare logic does not allow to prove that a guard fails or a procedure is undefined. Quite the opposite, it is designed to ensures that at most a guard in F can fail and that no procedure is undefined. To exclude certain final states we introduce the auxiliary predicate $\Gamma \vdash \langle c,s\rangle \Rightarrow\notin T$. Execution of program c in initial state s does not lead to a final state in T:

Definition 3.7 ►

$$\Gamma \vdash \langle c,s\rangle \Rightarrow\notin T \equiv \forall t.\ \Gamma \vdash \langle c,s\rangle \Rightarrow t \longrightarrow t \notin T$$

The most general triple for command c is:

$$\begin{array}{l}\Gamma\vdash_{/F} \{s.\ s = Z \wedge \Gamma\vdash \langle c,Normal\ s\rangle \Rightarrow\notin\{Stuck\} \cup Fault\ `\ (-\ F)\}\\ \quad c\\ \quad \{t.\ \Gamma\vdash \langle c,Normal\ Z\rangle \Rightarrow Normal\ t\},\{t.\ \Gamma\vdash \langle c,Normal\ Z\rangle \Rightarrow Abrupt\ t\}.\end{array}$$

The initial state is fixed to the auxiliary variable Z so that we can refer to it in the postcondition. Moreover, all initial state that can lead to a *Stuck* or *Fault* final state not in F are excluded by the precondition. The *Normal* final states belong to the postcondition for normal termination and all *Abrupt* final states to the postcondition for abrupt termination.

◀ Lemma 3.9 *MGT implies completeness*

Provided that the most general triple is derivable within the Hoare logic:

$$\begin{array}{l}\forall Z.\ \Gamma\vdash_{/F} \{s.\ s = Z \wedge \Gamma\vdash \langle c,Normal\ s\rangle \Rightarrow\notin\{Stuck\} \cup Fault\ `\ (-\ F)\}\\ \quad c\\ \quad \{t.\ \Gamma\vdash \langle c,Normal\ Z\rangle \Rightarrow Normal\ t\},\{t.\ \Gamma\vdash \langle c,Normal\ Z\rangle \Rightarrow Abrupt\ t\},\end{array}$$

then every valid triple $\Gamma\models_{/F} P\ c\ Q,A$ *is derivable in the Hoare logic:* $\Gamma\vdash_{/F} P\ c\ Q,A$.

Proof. We can derive the triple $\Gamma\vdash_{/F} P\ c\ Q,A$ from the MGT by using the consequence rule ConseqAux (cf. p. 44). Under the assumption of a $s \in P$, we instantiate the auxiliary variable Z in the MGT with s. From validity of the triple $\Gamma\models_{/F} P\ c\ Q,A$ we know that the execution of c does not end up in a *Stuck* or *Fault* state not in F. Thus s satisfies the precondition of the MGT. From the postcondition of the MGT we obtain a *Normal* or *Abrupt* final state t for the execution started in s. Since we know validity of the triple and have $s \in P$ we can conclude that $t \in Q$ or $t \in A$, respectively. □

We prove that the most general triple is derivable in the Hoare logic in two steps. First, we prove that the MGT is derivable under the assumption that the MGT of all procedures is derivable, and in the second step we discharge this assumption by the CallRec Rule for procedure calls.

◀ Lemma 3.10

Provided that the MGT for all procedures in Γ *is derivable:*

$$\begin{array}{l}\forall p{\in}dom\ \Gamma.\\ \quad \forall Z.\ \Gamma,\Theta\vdash_{/F}\\ \qquad \{s.\ s = Z \wedge \Gamma\vdash \langle Call\ p,Normal\ s\rangle \Rightarrow\notin\{Stuck\} \cup Fault\ `\ (-\ F)\}\\ \qquad Call\ p\\ \qquad \{t.\ \Gamma\vdash \langle Call\ p,Normal\ Z\rangle \Rightarrow Normal\ t\},\{t.\ \Gamma\vdash \langle Call\ p,Normal\ Z\rangle \Rightarrow Abrupt\ t\},\end{array}$$

then the MGT for a command c *is also derivable:*

$$\begin{array}{l}\forall Z.\ \Gamma,\Theta\vdash_{/F}\\ \quad \{s.\ s = Z \wedge \Gamma\vdash \langle c,Normal\ s\rangle \Rightarrow\notin\{Stuck\} \cup Fault\ `\ (-\ F)\}\\ \quad c\\ \quad \{t.\ \Gamma\vdash \langle c,Normal\ Z\rangle \Rightarrow Normal\ t\},\{t.\ \Gamma\vdash \langle c,Normal\ Z\rangle \Rightarrow Abrupt\ t\}.\end{array}$$

Proof. By induction on the syntax of command c.

Cases *Skip*, *Basic f*, *Spec r*, *Throw*: The MGTs are directly derivable by the corresponding Hoare logic rules.

Case *Call p*: The MGT is derived from the general assumption that all MGTs for procedures in *dom* Γ are derivable. Since the precondition ensures that execution does not get stuck we can conclude that $p \in dom\ \Gamma$.

Cases *Seq* c_1 c_2, *Cond* b c_1 c_2, *Catch* c_1 c_2, *DynCom* c_s: The MGTs can be directly derived from the corresponding Hoare logic rules, after adapting the MGTs for the components with the consequence rule. As an example, for sequential composition the induction hypothesis provides the MGT for c_1 and c_2:

$$\begin{array}{ll}\forall Z.\ \Gamma,\Theta\vdash_{/F} & \\ \quad \{s.\ s = Z \wedge \Gamma\vdash \langle c_1, Normal\ s\rangle \Rightarrow\notin \{Stuck\} \cup Fault\ `\ (-\ F)\} & \\ \quad c_1 & (*)\\ \quad \{t.\ \Gamma\vdash \langle c_1, Normal\ Z\rangle \Rightarrow Normal\ t\}, \{t.\ \Gamma\vdash \langle c_1, Normal\ Z\rangle \Rightarrow Abrupt\ t\} & \\ \forall Z.\ \Gamma,\Theta\vdash_{/F} & \\ \quad \{s.\ s = Z \wedge \Gamma\vdash \langle c_2, Normal\ s\rangle \Rightarrow\notin \{Stuck\} \cup Fault\ `\ (-\ F)\} & \\ \quad c_2 & (**)\\ \quad \{t.\ \Gamma\vdash \langle c_2, Normal\ Z\rangle \Rightarrow Normal\ t\}, \{t.\ \Gamma\vdash \langle c_2, Normal\ Z\rangle \Rightarrow Abrupt\ t\}. & \end{array}$$

We have to show the MGT for *Seq* c_1 c_2:

$$\begin{array}{l}\Gamma,\Theta\vdash_{/F}\\ \quad \{s.\ s = Z \wedge \Gamma\vdash \langle Seq\ c_1\ c_2, Normal\ s\rangle \Rightarrow\notin \{Stuck\} \cup Fault\ `\ (-\ F)\}\\ \quad Seq\ c_1\ c_2\\ \quad \{t.\ \Gamma\vdash \langle Seq\ c_1\ c_2, Normal\ Z\rangle \Rightarrow Normal\ t\}, \{t.\ \Gamma\vdash \langle Seq\ c_1\ c_2, Normal\ Z\rangle \Rightarrow Abrupt\ t\}.\end{array}$$

We can directly prove this with the SEQ Rule after adapting (*) and (**) by the consequence rule:

- $\Gamma,\Theta\vdash_{/F}$
  $\{s.\ s = Z \wedge \Gamma\vdash \langle Seq\ c_1\ c_2, Normal\ s\rangle \Rightarrow\notin \{Stuck\} \cup Fault\ `\ (-\ F)\}$
  c_1
  $\{t.\ \Gamma\vdash \langle c_1, Normal\ Z\rangle \Rightarrow Normal\ t \wedge \Gamma\vdash \langle c_2, Normal\ t\rangle \Rightarrow\notin \{Stuck\} \cup Fault\ `\ (-\ F)\}$,
  $\{t.\ \Gamma\vdash \langle Seq\ c_1\ c_2, Normal\ Z\rangle \Rightarrow Abrupt\ t\}$

- $\Gamma,\Theta\vdash_{/F}$
  $\{t.\ \Gamma\vdash \langle c_1, Normal\ Z\rangle \Rightarrow Normal\ t \wedge \Gamma\vdash \langle c_2, Normal\ t\rangle \Rightarrow\notin \{Stuck\} \cup Fault\ `\ (-\ F)\}$
  c_2
  $\{t.\ \Gamma\vdash \langle Seq\ c_1\ c_2, Normal\ Z\rangle \Rightarrow Normal\ t\}$,
  $\{t.\ \Gamma\vdash \langle Seq\ c_1\ c_2, Normal\ Z\rangle \Rightarrow Abrupt\ t\}$

From the precondition of the MGT for statement *Seq* c_1 c_2 we know that execution of *Seq* c_1 c_2 does not lead to a *Stuck* or *Fault* state not in F. This knowledge is broken down to the sub-statements and transfered to the pre- and postconditions with the consequence rule in order to link the derivations of c_1 and c_2 together.

Case *Guard f g c*: For $f \in F$ and according to the GUARANTEE Rule we do not have to show that guard g holds. Hence the MGT immediately follows from the MGT for c and the consequence rule. In the other case we have to show that the guard holds. We can exploit the precondition of the MGT of *Guard f g c*, which ensures that execution does not end up in a *Fault* state and thus the guard must hold.

Case *While b c*: As induction hypothesis we can assume the MGT for the loop body c:

$$\begin{array}{ll}\forall Z.\ \Gamma,\Theta\vdash_{/F} & \\ \quad \{s.\ s = Z \wedge \Gamma\vdash \langle c, Normal\ s\rangle \Rightarrow\notin \{Stuck\} \cup Fault\ `\ (-\ F)\} & \\ \quad c & (*)\\ \quad \{t.\ \Gamma\vdash \langle c, Normal\ Z\rangle \Rightarrow Normal\ t\}, \{t.\ \Gamma\vdash \langle c, Normal\ Z\rangle \Rightarrow Abrupt\ t\}. & \end{array}$$

We have to show the MGT for *While b c*:

$$\begin{array}{ll}\Gamma,\Theta\vdash_{/F} & \\ \quad \{s.\ s = Z \wedge \Gamma\vdash \langle While\ b\ c, Normal\ s\rangle \Rightarrow\notin \{Stuck\} \cup Fault\ `\ (-\ F)\} & \\ \quad While\ b\ c & (**)\\ \quad \{t.\ \Gamma\vdash \langle While\ b\ c, Normal\ Z\rangle \Rightarrow Normal\ t\}, \{t.\ \Gamma\vdash \langle While\ b\ c, Normal\ Z\rangle \Rightarrow Abrupt\ t\}. & \end{array}$$

To use the While Rule (cf. p. 40) we have to find an invariant that is strong enough to imply the MGT (∗∗) for *While b c*, but still can be derived from the MGT (∗) of the loop body c. To be more precise, we introduce the assertions P' and A' such that $\forall Z.\ \Gamma,\Theta\vdash (P'\ Z)\ \mathit{While}\ b\ c\ (P'\ Z \cap -b),(A'\ Z)$ implies (∗∗), and (∗) implies $\Gamma,\Theta\vdash (P'\ Z \cap b)\ c\ (P'\ Z),(A'\ Z)$:

- $\mathit{unroll} \equiv \{(s, t).\ s \in b \wedge \Gamma\vdash \langle c,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Normal}\ t\}^*$
- $P' \equiv$
  $\lambda Z.\ \{t.\ (Z, t) \in \mathit{unroll} \wedge$
  $\quad(\forall s_1.\ (Z, s_1) \in \mathit{unroll} \longrightarrow$
  $\quad\quad s_1 \in b \longrightarrow$
  $\quad\quad \Gamma\vdash \langle c,\mathit{Normal}\ s_1\rangle \Rightarrow\notin\{\mathit{Stuck}\} \cup \mathit{Fault}\ `\ (-\ F) \wedge$
  $\quad\quad (\forall s_2.\ \Gamma\vdash \langle c,\mathit{Normal}\ s_1\rangle \Rightarrow \mathit{Abrupt}\ s_2 \longrightarrow$
  $\quad\quad\quad \Gamma\vdash \langle \mathit{While}\ b\ c,\mathit{Normal}\ Z\rangle \Rightarrow \mathit{Abrupt}\ s_2))\}$
- $A' \equiv \lambda Z.\ \{t.\ \Gamma\vdash \langle \mathit{While}\ b\ c,\mathit{Normal}\ Z\rangle \Rightarrow \mathit{Abrupt}\ t\}$.

The relation *unroll* is the reflexive transitive closure of a single execution of the loop body from a state that satisfies the loop condition. It characterises the intermediate states that are reached by unrolling the loop. The intuition of the auxiliary variable Z is the initial state of the loop. The invariant P' captures three aspects. Firstly, every intermediate state t is reachable by unrolling the loop. Secondly, for every state s_1 that is reachable by unrolling the loop and for which the loop condition still holds, execution of the loop body does not get *Stuck* or lead to a *Fault* not in F. And thirdly, for every such state s_1, in case execution of the loop body terminates abruptly then this also terminates the whole loop. Note that the second and third aspect are general invariants for the reachable states of *unroll*. They are not restricted to the current state t.

1. (∗) implies $\Gamma,\Theta\vdash (P'\ Z \cap b)\ c\ (P'\ Z),(A'\ Z)$:
Z is the initial state of the whole loop. We apply the consequence rule ConseqAux. Thus our overall assumption is to have an intermediate state s, such that:

$$s \in P'\ Z \cap b. \qquad (\ast\ast\ast)$$

Therefore we know that s is reachable by unrolling: $(Z, s) \in \mathit{unroll}$, all the general properties of the reachable states, and that $s \in b$. We instantiate the auxiliary variable Z of MGT (∗) with this state s. The precondition of (∗) is a direct consequence of (∗∗∗) according to the second aspect of $P'\ Z$. For normal termination (∗) yields a *Normal* sate t from execution of the loop body: $\Gamma\vdash \langle c,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Normal}\ t$. We have to establish the invariant $P'\ Z$ for it. Since we know from (∗∗∗) that the invariant holds for s, we can guarantee it for t by unrolling the loop once more. For abrupt termination we have to show that this also exits the complete loop, which is the third aspect of the invariant (∗∗∗).

2. $\forall Z.\ \Gamma,\Theta\vdash (P'\ Z)\ \mathit{While}\ b\ c\ (P'\ Z \cap -b),(A'\ Z)$ implies (∗∗):
Again we start with the consequence rule ConseqAux and instantiate Z with Z which is identified with the initial state s by (∗∗). The precondition of (∗∗) gives us the general assumption:

$$\Gamma\vdash \langle \mathit{While}\ b\ c,\mathit{Normal}\ Z\rangle \Rightarrow\notin\{\mathit{Stuck}\} \cup \mathit{Fault}\ `\ (-\ F). \qquad (\ast\ast\ast)$$

To establish the preconditon we have to prove the invariant for state Z: $Z \in P'\ Z$. Since *unroll* is reflexive, $(Z, Z) \in \mathit{unroll}$ is trivial. The general properties about all unrolled states are proven by reflexive transitive closure induction on *unroll*, under the assumption of $(***)$.

To establish the postcondition of $(**)$ for normal termination we have to show that all $t \in P'\ Z \cap -\ b$ are also proper final states of the execution of the while loop: $\Gamma\vdash \langle \mathit{While}\ b\ c, \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Normal}\ t$. We only need $(Z, t) \in \mathit{unroll}$ and $t \notin b$ in order to prove this, again by reflexive transitive closure induction on *unroll*. The postcondition for abrupt termination of $(**)$ and $A'\,Z$ are the same, so there is nothing to do in this case. □

The next lemma states that the MGT for all procedures in Γ is derivable:

Lemma 3.11 ▸

$$\begin{array}{l}\forall p{\in}\mathit{dom}\ \Gamma.\\ \quad \forall Z.\ \Gamma\vdash_{/F} \{s.\ s = Z \wedge \Gamma\vdash \langle \mathit{Call}\ p, \mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\} \cup \mathit{Fault}\ `\ (-\ F)\}\\ \qquad\qquad (\mathit{Call}\ p)\\ \qquad\qquad \{t.\ \Gamma\vdash \langle \mathit{Call}\ p, \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Normal}\ t\}, \{t.\ \Gamma\vdash \langle \mathit{Call}\ p, \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Abrupt}\ t\}\end{array}$$

Proof. With the CallRec Rule we augment the context with all the specifications for all $p \in \mathit{dom}\ \Gamma$ and all Z.

$$\begin{array}{l}\mathit{Specs} =\\ (\bigcup_{p\in \mathit{dom}\ \Gamma}\\ \quad \bigcup_Z \{(\{s.\ s = Z \wedge \Gamma\vdash \langle \mathit{Call}\ p, \mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\} \cup \mathit{Fault}\ `\ (-\ F)\},\\ \qquad\quad p,\\ \qquad\quad \{t.\ \Gamma\vdash \langle \mathit{Call}\ p, \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Normal}\ t\},\\ \qquad\quad \{t.\ \Gamma\vdash \langle \mathit{Call}\ p, \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Abrupt}\ t\})\})\end{array}$$

In this augmented context we can derive the MGT for the procedure bodies by Lemma 3.10, since we can provide the MGTs for the calls by the assumption rule:

$$\begin{array}{l}\forall Z.\ \Gamma,\mathit{Specs}\vdash_{/F} \{s.\ s = Z \wedge \Gamma\vdash \langle \mathit{the}\ (\Gamma\ p), \mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\} \cup \mathit{Fault}\ `\ (-\ F)\}\\ \qquad\qquad \mathit{the}\ (\Gamma\ p)\\ \qquad\qquad \{t.\ \Gamma\vdash \langle \mathit{the}\ (\Gamma\ p), \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Normal}\ t\},\\ \qquad\qquad \{t.\ \Gamma\vdash \langle \mathit{the}\ (\Gamma\ p), \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Abrupt}\ t\}\end{array}$$

Since execution of a defined procedure is reduced to the execution of its body we can adapt this MGT to:

$$\begin{array}{l}\forall Z.\ \Gamma,\mathit{Specs}\vdash_{/F} \{s.\ s = Z \wedge \Gamma\vdash \langle \mathit{Call}\ p, \mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\} \cup \mathit{Fault}\ `\ (-\ F)\}\\ \qquad\qquad \mathit{the}\ (\Gamma\ p)\\ \qquad\qquad \{t.\ \Gamma\vdash \langle \mathit{Call}\ p, \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Normal}\ t\},\\ \qquad\qquad \{t.\ \Gamma\vdash \langle \mathit{Call}\ p, \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Abrupt}\ t\}\end{array}$$

As this matches to the premise of the CallRec Rule we have finshed the proof. □

Lemma 3.11 allows to discharge the assumption of Lemma 3.10. Therefore we have proven that the MGT is derivable in the Hoare logic. With Lemma 3.9 we arrive at the completeness theorem:

Theorem 3.12 ▸
Completeness

$$\Gamma\models_{/F} P\ c\ Q,A \longrightarrow \Gamma\vdash_{/F} P\ c\ Q,A$$

For soundness we also have a version which takes the context Θ into account, but for completeness this does not work. If there is a malicious specification in context Θ every triple is regarded as valid in the extended notion of validity. In a sense the Hoare logic is “too correct”. It does not allow to derive arbitrary nonsense from a false specification in the context. Initially we can only derive nonsense about the malicious procedure itself. Of course this can be expanded to a program that uses this procedure. However, for statements that do not refer to the procedure we still can only derive sound triples. This stems from the fact that the Hoare logic rules work strictly syntax directed. Only if we already have a proven fact about a statement we may adapt it by the consequence rule. By adding the following semantical rule to the Hoare logic we can remedy this situation:

$$\frac{\forall n.\ \Gamma,\Theta \overset{n}{\models}_{/F} P\ c\ Q,A \qquad \neg\ \Gamma \models_{/F} P\ c\ Q,A}{\Gamma,\Theta \vdash_{/F} P\ c\ Q,A}\ (\textsc{ExFalso})$$

If a triple is valid under context Θ but invalid without the context, then nevertheless we can derive it. This rule is trivially sound and does not contribute to pure completeness, since we can only derive invalid triples with it. Therefore the completeness Theorem 3.12 still makes sense and we have not just defined completeness into the Hoare logic. Note that we use the more restrictive $\forall n.\ \Gamma,\Theta \overset{n}{\models}_{/F} P\ c\ Q,A$ instead of $\Gamma,\Theta \models_{/F} P\ c\ Q,A$ in the premise of the ExFalso Rule, so that the proof of lemma 3.6 still works. With the ExFalso Rule we can extend the completeness theorem to work with any context:

◂ Lemma 3.13 *Completeness within context*

If we augment the Hoare logic with rule ExFalso *then:*

$$(\forall n.\ \Gamma,\Theta \overset{n}{\models}_{/F} P\ c\ Q,A) \longrightarrow \Gamma,\Theta \vdash_{/F} P\ c\ Q,A.$$

Why is such a lemma useful at all? In the end we are only interested in deriving triples out of an empty context, so Theorem 3.12 is sufficient. The answer is "proof engineering". Although the Hoare calculus is complete there are a bunch of rules that are practically useful, but not derivable from the given set of rules. For example, all semantic preserving transformations of the program. Such rules can be proven semantically sound and the completeness theorem brings them into the Hoare logic. Those rules are only applicable in an empty context. However, with Lemma 3.13 it is also possible to make those rules available in arbitrary contexts. Of course it is also possible to extend the core calculus with all the desired rules. However, this results in a rather monolithic structure of the theories, whereas Lemma 3.13 allows a modular and incremental development. This approach also allows to introduce rules to the Hoare calculus that would be rejected by the HOL type system, if they are directly inserted to the inductive definition of the Hoare logic. For example, in previous versions of the Hoare logic the consequence rule ConseqAux was part of the core rules, instead of the Conseq Rule. As explained in Section 3.1.1 this has the odd effect that the type of the auxiliary variable is fixed. However, for the completeness proof we only need the auxiliary variables to fix the state. Therefore it is sufficient to fix the type of the auxiliary variable to the state space type. Then the completeness proof can be done with this restricted version of the consequence rule. Afterwards the polymorphic variant of the consequence rule can be proven sound and inserted to the calculus by the completeness theorem.

3.2 Total Correctness

Total correctness means partial correctness plus termination. This is directly reflected in the validity notion for total correctness:

Definition 3.8 ► *Validity (total correctness)*

$$\Gamma \models_{t/F} P\ c\ Q{,}A \equiv \Gamma \models_{/F} P\ c\ Q{,}A \wedge (\forall s {\in} Normal\ `\ P.\ \Gamma \vdash c \downarrow s)$$

The various judgements for total correctness are distinguished from partial correctness by the subscript $_t$.

Definition 3.9 ► *Hoare logic for Simpl (total correctness)*

The total correctness Hoare logic for Simpl is inductively defined by the rules in Figure 3.3. The judgement has the following form:

$$\Gamma{,}\Theta \vdash_{t/F} P\ c\ Q{,}A,$$

where:

$$\begin{array}{llll}
\Gamma :: 'p \rightharpoonup ('s,\ 'p,\ 'f)\ com & & P{,}Q{,}A :: 's\ set \\
\Theta :: ('s\ set \times 'p \times 's\ set \times 's\ set)\ set & & c :: ('s,\ 'p,\ 'f)\ com \\
F :: 'f\ set & &
\end{array}$$

$$\frac{}{\Gamma{,}\Theta \vdash_{t/F} Q\ Skip\ Q{,}A}\ (\textsc{Skip}) \qquad \frac{}{\Gamma{,}\Theta \vdash_{t/F} \{s.\ f\ s \in Q\}\ (Basic\ f)\ Q{,}A}\ (\textsc{Basic})$$

$$\frac{\Gamma{,}\Theta \vdash_{t/F} P\ c_1\ R{,}A \qquad \Gamma{,}\Theta \vdash_{t/F} R\ c_2\ Q{,}A}{\Gamma{,}\Theta \vdash_{t/F} P\ (Seq\ c_1\ c_2)\ Q{,}A}\ (\textsc{Seq})$$

$$\frac{\Gamma{,}\Theta \vdash_{t/F} (P \cap b)\ c_1\ Q{,}A \qquad \Gamma{,}\Theta \vdash_{t/F} (P \cap -b)\ c_2\ Q{,}A}{\Gamma{,}\Theta \vdash_{t/F} P\ (Cond\ b\ c_1\ c_2)\ Q{,}A}\ (\textsc{Cond})$$

$$\frac{wf\ r \qquad \forall \sigma.\ \Gamma{,}\Theta \vdash_{t/F} (\{\sigma\} \cap P \cap b)\ c\ (\{t.\ (t,\ \sigma) \in r\} \cap P){,}A}{\Gamma{,}\Theta \vdash_{t/F} P\ (While\ b\ c)\ (P \cap -b){,}A}\ (\textsc{While})$$

$$\frac{\begin{array}{c}(P,\ p,\ Q,\ A) \in Specs \\ wf\ r \qquad Specs\text{-}wf = (\lambda p\ \sigma.\ (\lambda(P,\ q,\ Q,\ A).\ (P \cap \{s.\ ((s,\ q),\ (\sigma,\ p)) \in r\},\ q,\ Q,\ A))\ `\ Specs) \\ \forall (P,\ p,\ Q,\ A){\in}Specs.\ p \in dom\ \Gamma \wedge (\forall \sigma.\ \Gamma{,}\Theta \cup Specs\text{-}wf\ p\ \sigma {\vdash_{t/F}} (\{\sigma\} \cap P)\ (the\ (\Gamma\ p))\ Q{,}A)\end{array}}{\Gamma{,}\Theta \vdash_{t/F} P\ (Call\ p)\ Q{,}A}\ (\textsc{CallRec})$$

$$\frac{\Gamma{,}\Theta \vdash_{t/F} (g \cap P)\ c\ Q{,}A}{\Gamma{,}\Theta \vdash_{t/F} (g \cap P)\ (Guard\ f\ g\ c)\ Q{,}A}\ (\textsc{Guard}) \qquad \frac{f \in F \qquad \Gamma{,}\Theta \vdash_{t/F} (g \cap P)\ c\ Q{,}A}{\Gamma{,}\Theta \vdash_{t/F} P\ (Guard\ f\ g\ c)\ Q{,}A}\ (\textsc{Guarantee})$$

$$\frac{}{\Gamma{,}\Theta \vdash_{t/F} A\ Throw\ Q{,}A}\ (\textsc{Throw}) \qquad \frac{\Gamma{,}\Theta \vdash_{t/F} P\ c_1\ Q{,}R \qquad \Gamma{,}\Theta \vdash_{t/F} R\ c_2\ Q{,}A}{\Gamma{,}\Theta \vdash_{t/F} P\ (Catch\ c_1\ c_2)\ Q{,}A}\ (\textsc{Catch})$$

$$\frac{}{\Gamma{,}\Theta \vdash_{t/F} \{s.\ (\forall t.\ (s,\ t) \in r \longrightarrow t \in Q) \wedge (\exists t.\ (s,\ t) \in r)\}\ (Spec\ r)\ Q{,}A}\ (\textsc{Spec})$$

$$\frac{\forall s{\in}P.\ \Gamma{,}\Theta \vdash_{t/F} P\ (c_s\ s)\ Q{,}A}{\Gamma{,}\Theta \vdash_{t/F} P\ (DynCom\ c_s)\ Q{,}A}\ (\textsc{DynCom})$$

. . . .

$$\frac{\forall s{\in}P.\ \exists P'\ Q'\ A'.\ \Gamma{,}\Theta \vdash_{t/F} P'\ c\ Q'{,}A' \wedge s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A}{\Gamma{,}\Theta \vdash_{t/F} P\ c\ Q{,}A}\ (\textsc{Conseq}) \qquad \frac{(P,\ p,\ Q,\ A) \in \Theta}{\Gamma{,}\Theta \vdash_{t/F} P\ (Call\ p)\ Q{,}A}\ (\textsc{Asm})$$

Figure 3.3: Hoare logic for total correctness

Most of the rules are structurally equivalent to their partial correctness counterparts, except for the rules for loops and recursive procedures. The central idea is that termination is ensured by a well-founded relation on the state space. A relation r is well-founded if and only if there is no infinite descending chain :

$$\ldots, (s_3, s_2), (s_2, s_1), (s_1, s_0) \in r$$

The WHILE Rule fixes the initial state of the loop body with σ. After the loop is executed the final state s has to be smaller with respect to the well-founded relation r. The predicate *wf r* expresses that r is well-founded. The PROCREC Rule for recursive procedures has the same structure. The preconditions of the procedure specifications in the context are restricted to smaller states. Again with respect to a well-founded relation r. In this case the relation does not only depend on the state space but also on the procedure name. This is useful to handle mutually recursive procedures and crucial in the completeness proof.

3.2.1 Soundness

Soundness for total correctness ensures that every derived Hoare triple is indeed partially correct and that the program terminates. The basic proof structure is again induction on the Hoare logic derivation. Compared to the partial correctness proof we do not need to argue on the depth of recursion this time. Since the Rules WHILE and CALLREC are equipped with a well-founded relation we can instead use induction on this relation. First, we introduce the notion of validity within a context:

$$\Gamma,\Theta \models_{t/F} P\ c\ Q,A \equiv (\forall (P, p, Q, A) \in \Theta.\ \Gamma \models_{t/F} P\ (Call\ p)\ Q,A) \longrightarrow \Gamma \models_{t/F} P\ c\ Q,A$$

◂ Definition 3.10 *Validity within context*

Now we can proceed with the main lemma, soundness within a context:

$$\Gamma,\Theta \vdash_{t/F} P\ c\ Q,A \longrightarrow \Gamma,\Theta \models_{t/F} P\ c\ Q,A$$

◂ Lemma 3.14 *Soundness within context*

Proof. By induction on the Hoare logic derivation. Since the Hoare logic, the operational semantics, and the termination judgement $\Gamma \vdash c \downarrow s$ all follow the same structure the proof is straightforward in most cases. The interesting cases are WHILE and CALLREC.

Case WHILE. As induction hypothesis we get a well-founded relation r and validity of the loop body c:

- *wf r* (∗)
- $\forall \sigma.\ \Gamma,\Theta \models_{t/F} (\{\sigma\} \cap P \cap b)\ c\ (\{\tau.\ (\tau, \sigma) \in r\} \cap P),A.$ (∗∗)

We have to show validity of the whole loop:

$$\Gamma,\Theta \models_{t/F} P\ (While\ b\ c)\ (P \cap -b),A.$$

According to the definition of total correctness we have to show partial correctness and termination.

Partial correctness: By unfolding Definition 3.3 of partial correctness within a context we can assume:

- $\forall (P, p, Q, A) \in \Theta.\ \Gamma \models_{t/F} P\ Call\ p\ Q,A,$
- $\Gamma \vdash \langle While\ b\ c, Normal\ s\rangle \Rightarrow t,$

- $s \in P$, and
- $t \notin \mathit{Fault}\ `\ F$,

and have to prove that the postcondition holds for t:

$$t \in \mathit{Normal}\ `\ (P \cap -b) \cup \mathit{Abrupt}\ `\ A.$$

We exploit (∗) and do a well-founded induction on the *initial* state s. This gives us an induction hypothesis for all executions of *While b c* in a smaller state than s. In case $s \notin b$ we exit the loop and can immediately finish the proof. Otherwise we first execute the loop body c and reach an intermediate state τ. From validity (∗∗) we know that state τ is smaller according to relation r: $(\tau, s) \in r$. Moreover, we know that the invariant holds at state τ. If $\tau \notin b$ then $\tau = t$ and we are finished. Otherwise we complete the proof with the induction hypothesis since τ is a smaller state than s and therefore the recursive execution of *While b c* leads to a proper final state.

Termination: By unfolding the definition of total correctness within a context we can assume:

- $\forall (P, p, Q, A){\in}\Theta.\ \Gamma{\models_{t/F}}\ P\ \mathit{Call}\ p\ Q{,}A$ and
- $s \in P$

and have to prove termination of the loop:

$$\Gamma{\vdash}\mathit{While}\ b\ c \downarrow \mathit{Normal}\ s.$$

Again we exploit (∗) and do a well-founded induction on the *initial* state s. This gives us termination of *While b c* started in a *Normal* state smaller state than s. In case $s \notin b$ we exit the loop and can immediately finish the proof. Otherwise validity (∗∗) yields termination of the loop body. The execution of c yields an intermediate state τ. From validity (∗∗) we know that state τ is smaller according to relation r: $(\tau, s) \in r$. If $\tau \notin b$ then the loop terminates immediately. Otherwise we complete the proof with the induction hypothesis since τ is a smaller state than s and therefore the recursive execution of *While b c* terminates.

Case CallRec. From the induction we get the following hypotheses:

- $(P, p, Q, A) \in \mathit{Specs}$
- $\mathit{wf}\ r$ (∗)
- $\mathit{Specs\text{-}wf} = (\lambda p\ \sigma.\ (\lambda(P, q, Q, A).\ (P \cap \{s.\ ((s, q), (\sigma, p)) \in r\}, q, Q, A))\ `\ \mathit{Specs})$
- $\forall (P, p, Q, A){\in}\mathit{Specs}.$ (∗∗)
  $p \in \mathit{dom}\ \Gamma \wedge (\forall \sigma.\ \Gamma{,}\Theta \cup \mathit{Specs\text{-}wf}\ p\ \sigma{\models_{t/F}}\ (\{\sigma\} \cap P)\ (\mathit{the}\ (\Gamma\ p))\ Q{,}A)$

We have to show validity of the procedure call within context Θ:

$$\Gamma{,}\Theta{\models_{t/F}}\ P\ (\mathit{Call}\ p)\ Q{,}A.$$

By expanding the definition of validity within context Θ we have to show:

$$(\forall (P, p, Q, A){\in}\Theta.\ \Gamma{\models_{t/F}}\ P\ (\mathit{Call}\ p)\ Q{,}A) \longrightarrow \Gamma{\models_{t/F}}\ P\ (\mathit{Call}\ p)\ Q{,}A.$$

The central idea is to exploit hypothesis (**) to get hold of the validity for the procedure body: $\forall\sigma.\ \Gamma\models_{t/F}(\{\sigma\}\cap P)\ (\mathit{the}\ (\Gamma\ p))\ Q,A$. Therefore we have to discharge the context $\Theta \cup \mathit{Specs\text{-}wf}\ p\ \sigma$. Once the validity of the procedure body is established the validity of the corresponding procedure call is a direct consequence from the operational semantics and termination. For the specifications in Θ we can already assume validity, since we show validity of the procedure call in this context. To discharge the specifications in *Specs-wf*, we use well-founded induction for the following generalised goal:

We assume:

- $\forall(P, p, Q, A)\in\Theta.\ \Gamma\models_{t/F} P\ (\mathit{Call}\ p)\ Q,A$ and (∗∗∗)
- $(P, p, Q, A) \in \mathit{Specs}$

and show:

$$\Gamma\models_{t/F}(\{\sigma\}\cap P)\ (\mathit{the}\ (\Gamma\ p))\ Q,A.$$

We use (∗) and do a well-founded induction on the initial configuration (σ, p). Therefore we get an induction hypothesis for all (s, q), such that $((s, q), (\sigma, p)) \in r$. Hence with the definition of validity we have:

$$\forall(P', q, Q', A')\in\mathit{Specs}.\ \Gamma\models_{t/F}(P'\cap\{s.\ ((s, q), (\sigma, p))\in r\})\ (\mathit{the}\ (\Gamma\ q))\ Q',A'.$$

This is lifted from the body to the corresponding call by using the operational semantics and termination:

$$\forall(P', q, Q', A')\in\mathit{Specs}.\ \Gamma\models_{t/F}(P'\cap\{s.\ ((s, q), (\sigma, p))\in r\})\ (\mathit{Call}\ q)\ Q',A'.$$

This is exactly the definition of *Specs-wf*. Together with (∗∗∗) we can discharge the context of (∗∗) and have proven validity of the body. □

The direct consequence of this lemma is the pure soundness theorem for total correctness:

$$\Gamma\vdash_{t/F} P\ c\ Q,A \longrightarrow \Gamma\models_{t/F} P\ c\ Q,A$$

◂ Theorem 3.15
Soundness

3.2.2 Completeness

The basic strategy to prove completeness for total correctness is the same as for partial correctness in section 3.1.4 and extends the work of Nipkow [77, 78] to Simpl. Again we define the notion of the most general triple which implies completeness. Since validity for total correctness ensures termination we add the $\Gamma\vdash c\downarrow \mathit{Normal}\ s$ to the precondition of the MGT. So for total correctness the most general triple for command c is:

$$\begin{array}{l}\Gamma\vdash_{t/F}\{s.\ s = Z \wedge \Gamma\vdash\langle c,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\}\cup \mathit{Fault}\ `\ (-\ F) \wedge \Gamma\vdash c\downarrow \mathit{Normal}\ s\}\\ \quad c\\ \quad \{t.\ \Gamma\vdash\langle c,\mathit{Normal}\ Z\rangle \Rightarrow \mathit{Normal}\ t\},\{t.\ \Gamma\vdash\langle c,\mathit{Normal}\ Z\rangle \Rightarrow \mathit{Abrupt}\ t\}\end{array}$$

The key difference to partial correctness is how to deal with loops and recursion. The Rules WHILE and CALLREC now demand a well-founded relation. In case of the loop it is straightforward to construct, but for the recursive procedures it gets rather involved. To satisfy the WHILE Rule, execution of the loop body has to decrease the state with respect to a well-founded relation r. Here $(t, s) \in r$ means that t is smaller than s. We consider a terminating loop. We regard those states as smaller that are "nearer" to the end of the computation. Hence the state after execution of the loop body is smaller than the initial state. That is the idea of the following relation:

Definition 3.11 ► $$(<^{\Gamma}_{b,c}) \equiv \{(t, s).\ \Gamma\vdash \mathit{While}\ b\ c \downarrow \mathit{Normal}\ s \wedge s \in b \wedge \Gamma\vdash \langle c,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Normal}\ t\}$$

To show well-foundedness of this relation, we use the lemma of the library that expresses that a relation is well-founded if and only if there is no infinite descending chain:

Lemma 3.16 ► $$\mathit{wf}\ r = (\nexists f.\ \forall i.\ (f\ (i + 1), f\ i) \in r)$$

An infinite descending chain is modelled as a function f from the natural numbers to the elements of the relation r, for which $f\ (i + 1)$ is smaller than $f\ i$. The relation is well-founded if there is no such function. By induction on the termination judgement we show that if *While* terminates then we eventually reach a state where the loop condition becomes false:

Lemma 3.17 ► *If* $\Gamma\vdash \mathit{While}\ b\ c \downarrow \mathit{Normal}\ (f\ k)$ *and* $\forall i.\ \Gamma\vdash \langle c,\mathit{Normal}\ (f\ i)\rangle \Rightarrow \mathit{Normal}\ (f\ (i + 1))$ *then* $\exists i.\ f\ i \notin b$.

The general $f\ k$ rather than the more intuitive $f\ 0$ is required by the inductive proof. Together with Lemma 3.16 we gain well-foundedness of $<^{\Gamma}_{b,c}$:

Lemma 3.18 ► $$\mathit{wf}\ (<^{\Gamma}_{b,c})$$

The termination ordering $<^{\Gamma}_{b,c}$ for the while loop was easy to define with the means of the big-step semantics, because of its uniform computation that consists of the execution of the body followed by the test of the condition. The "distance" between two states in $<^{\Gamma}_{b,c}$ is one execution of the loop body. In case of (mutually) recursive procedures the situation is different. There is no uniform code segment that is executed between every two procedure calls. For partial correctness the proof that the MGT is derivable is divided in two main lemmas (cf. Lemmas 3.10 and 3.11):

- Derive the MGT under the assumption that the MGT of all procedures in Γ is derivable.
- Derive the MGT for the procedures.

The proof of the second lemma builds on the first. With the rule CallRec (for partial correctness) the MGTs for all procedures are put into the context, so that they are trivially derivable by the assumption rule. In this augmented context the first lemma is applicable. For total correctness the second step is problematic. The rule CallRec (for total correctness) only allows to put restricted MGTs to the context. The specifications are only applicable to configurations that are smaller with respect to a well-founded relation. To get along with those restricted MGTs we need to find an appropriate well-founded relation. Intuitively we only need the MGTs for procedure calls in configurations that are reachable from the initial state. Since we also know that the computation terminates this relation should be well-founded. The problem is that our big-step semantics is too coarse-grained to express a "reachable configuration". It only relates initial to final states. To remedy this situation we introduce a small-step semantics. Then we define the relation of reachable configurations of a terminating computation and prove that it is indeed well-founded. Finally we have to show that it is sufficient to focus on the restricted set of MGTs to derive the MGT of a procedure call.

Termination ordering for (mutually) recursive procedure calls We define a small-step semantics for Simpl, in order to formulate that a procedure call is reachable from another one. This is the basic building block for the well-founded relation that we need for rule CallRec. A big-step semantic executes the whole program at once and relates the initial with the final states. A small-step semantics is a single step relation between configurations of the computation. Via the reflexive transitive closure we can express that a configuration is reachable from another one. The design principle of the small-step semantics is to make it easy to identify those configurations where a procedure call is executed next. Basically a configuration consists of a list of pending statements and the current state. The head of the statement list is the next command to be executed. Compound statements like c_1; c_2 are first decomposed to the list of components [c_1, c_2], until an atomic statement is the first one in the list. This statement is then executed and removed from the list. Moreover, to handle abrupt termination we keep track of a stack of so called *continuations*. A continuation consists of two statement lists, one for normal termination and one for abrupt termination. The continuation stack structures the computation into blocks. A statement *Catch* c_1 c_2 opens a new block, by pushing the pending statements to the stack. If the pending statements are completely processed the compoutation continues by popping the continuation stack. Depending on the current state the computation is continued with the statements for normal or abrupt termination, respectively.

◂ Definition 3.12

$$\begin{array}{lcl} ('s,'p,'f)\ \mathit{continuation} & = & ('s,'p,'f)\ \mathit{com\ list} \times ('s,'p,'f)\ \mathit{com\ list} \\ ('s,'p,'f)\ \mathit{config} & = & ('s,'p,'f)\ \mathit{com\ list} \times ('s,'p,'f)\ \mathit{continuation\ list} \times ('s,'f)\ \mathit{xstate} \end{array}$$

◂ Definition 3.13
Small-step semantics for Simpl

The operational small-step semantics: $\Gamma\vdash \langle cs, css, s\rangle \rightarrow \langle cs', css', t\rangle$, *is defined inductively by the rules in Figure 3.4. In procedure environment* Γ *a single computation step transforms configuration* $\langle cs, css, s\rangle$ *to* $\langle cs', css', t\rangle$, *where:*

$$\begin{array}{ll} \Gamma & :: 'p \rightharpoonup ('s, 'p, 'f)\ \mathit{com} \\ s, t & :: ('s, 'f)\ \mathit{xstate} \\ cs, cs' & :: ('s, 'p, 'f)\ \mathit{com\ list} \\ css, css' & :: ('s, 'p, 'f)\ \mathit{continuation\ list}. \end{array}$$

Moreover, we write $\Gamma\vdash \langle cs, css, s\rangle \rightarrow^+ \langle cs', css', t\rangle$ *and* $\Gamma\vdash \langle cs, css, s\rangle \rightarrow^* \langle cs', css', t\rangle$ *for the transitive and reflexive transitive closure of the single step computation.*

Compound statements are decomposed by augmenting the list of pending statements *cs* with the components. The state component of the configuration stays the same during this decomposition, until an atomic statement is reached. When a configuration $\langle$*Catch* c_1 c_2·*cs*,*css*,*Normal s*$\rangle$ is reached we enter a new block to execute c_1. In case of normal termination of c_1 the pending statements *cs* are executed, in case of abrupt termination the handler c_2 is inserted. Therefore the next configuration is $\langle[c_1],(cs,c_2\cdot cs)\cdot css,\mathit{Normal}\ s\rangle$. When the current block is completely processed, i.e. there are no more pending statements, the continuation stack is taken into account. If it is also empty we have reached the *final configuration*: $\langle[], [], s\rangle$. Otherwise the continuation is chosen according to the current state. In case of an *Abrupt* state the component for abrupt termination is selected, and in all other cases the one for normal termination is chosen. *Fault* and *Stuck* states skip the execution of the pending statements.

$$\frac{}{\Gamma\vdash \langle \mathit{Skip}\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle cs,\ css,\ \mathit{Normal}\ s\rangle}\ (\textsc{Skip}) \qquad \frac{}{\Gamma\vdash \langle \mathit{Basic}\ f\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle cs,\ css,\ \mathit{Normal}\ (f\ s)\rangle}\ (\textsc{Basic})$$

$$\frac{}{\Gamma\vdash \langle \mathit{Seq}\ c_1\ c_2\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle c_1\cdot c_2\cdot cs,\ css,\ \mathit{Normal}\ s\rangle}\ (\textsc{Seq})$$

$$\frac{s\in b}{\Gamma\vdash \langle \mathit{Cond}\ b\ c_1\ c_2\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle c_1\cdot cs,\ css,\ \mathit{Normal}\ s\rangle}\ (\textsc{CondTrue})$$

$$\frac{s\notin b}{\Gamma\vdash \langle \mathit{Cond}\ b\ c_1\ c_2\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle c_2\cdot cs,\ css,\ \mathit{Normal}\ s\rangle}\ (\textsc{CondFalse})$$

$$\frac{s\in b}{\Gamma\vdash \langle \mathit{While}\ b\ c\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle c\cdot \mathit{While}\ b\ c\cdot cs,\ css,\ \mathit{Normal}\ s\rangle}\ (\textsc{WhileTrue})$$

$$\frac{s\notin b}{\Gamma\vdash \langle \mathit{While}\ b\ c\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle cs,\ css,\ \mathit{Normal}\ s\rangle}\ (\textsc{WhileFalse})$$

$$\frac{\Gamma\ p = \lfloor bdy\rfloor}{\Gamma\vdash \langle \mathit{Call}\ p\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle [bdy],\ (cs,\ \mathit{Throw}\cdot cs)\cdot css,\ \mathit{Normal}\ s\rangle}\ (\textsc{Call})$$

$$\frac{\Gamma\ p = \mathit{None}}{\Gamma\vdash \langle \mathit{Call}\ p\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle cs,\ css,\ \mathit{Stuck}\rangle}\ (\textsc{CallUndefined})$$

$$\frac{s\in g}{\Gamma\vdash \langle \mathit{Guard}\ f\ g\ c\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle c\cdot cs,\ css,\ \mathit{Normal}\ s\rangle}\ (\textsc{Guard})$$

$$\frac{s\notin g}{\Gamma\vdash \langle \mathit{Guard}\ f\ g\ c\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle cs,\ css,\ \mathit{Fault}\ f\rangle}\ (\textsc{GuardFault})$$

$$\frac{}{\Gamma\vdash \langle \mathit{Throw}\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle cs,\ css,\ \mathit{Abrupt}\ s\rangle}\ (\textsc{Throw})$$

$$\frac{}{\Gamma\vdash \langle \mathit{Catch}\ c_1\ c_2\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle [c_1],\ (cs,\ c_2\cdot cs)\cdot css,\ \mathit{Normal}\ s\rangle}\ (\textsc{Catch})$$

$$\frac{(s,t)\in r}{\Gamma\vdash \langle \mathit{Spec}\ r\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle cs,\ css,\ \mathit{Normal}\ t\rangle}\ (\textsc{Spec}) \qquad \frac{\forall t.\ (s,t)\notin r}{\Gamma\vdash \langle \mathit{Spec}\ r\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle cs,\ css,\ \mathit{Stuck}\rangle}\ (\textsc{SpecStuck})$$

$$\frac{}{\Gamma\vdash \langle \mathit{DynCom}\ c\cdot cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle c\ s\cdot cs,\ css,\ \mathit{Normal}\ s\rangle}\ (\textsc{DynCom})$$

. . . .

$$\frac{}{\Gamma\vdash \langle c\cdot cs,\ css,\ \mathit{Fault}\ f\rangle \to \langle cs,\ css,\ \mathit{Fault}\ f\rangle}\ (\textsc{FaultProp}) \qquad \frac{}{\Gamma\vdash \langle c\cdot cs,\ css,\ \mathit{Stuck}\rangle \to \langle cs,\ css,\ \mathit{Stuck}\rangle}\ (\textsc{StuckProp})$$

$$\frac{}{\Gamma\vdash \langle c\cdot cs,\ css,\ \mathit{Abrupt}\ s\rangle \to \langle cs,\ css,\ \mathit{Abrupt}\ s\rangle}\ (\textsc{AbruptProp})$$

$$\frac{}{\Gamma\vdash \langle [],\ (nrms,\ abrs)\cdot css,\ \mathit{Normal}\ s\rangle \to \langle nrms,\ css,\ \mathit{Normal}\ s\rangle}\ (\textsc{ExitBlockNormal})$$

$$\frac{}{\Gamma\vdash \langle [],\ (nrms,\ abrs)\cdot css,\ \mathit{Abrupt}\ s\rangle \to \langle abrs,\ css,\ \mathit{Normal}\ s\rangle}\ (\textsc{ExitBlockAbrupt})$$

$$\frac{}{\Gamma\vdash \langle [],\ (nrms,\ abrs)\cdot css,\ \mathit{Fault}\ f\rangle \to \langle nrms,\ css,\ \mathit{Fault}\ f\rangle}\ (\textsc{ExitBlockFault})$$

$$\frac{}{\Gamma\vdash \langle [],\ (nrms,\ abrs)\cdot css,\ \mathit{Stuck}\rangle \to \langle nrms,\ css,\ \mathit{Stuck}\rangle}\ (\textsc{ExitBlockStuck})$$

Figure 3.4: Small-step semantics for Simpl

The only surprising rule is CALL. A more intuitive, simpler rule is:

$$\frac{\Gamma\ p = \lfloor bdy \rfloor}{\Gamma \vdash \langle \mathit{Call}\ p{\cdot}cs,\ css,\ \mathit{Normal}\ s\rangle \to \langle bdy{\cdot}cs,\ css,\ \mathit{Normal}\ s\rangle}\ (\textsc{CallSimple})$$

Instead I have decided to artificially open a new block. The rule itself is more complicated, but the target configuration is somehow simpler. Every procedure body starts its execution in a configuration where there are no pending statements. The reason for this more complicated rule lies in the original motivation for the small-step semantics: to prove well-foundedness of the reachable sequence of procedure calls of a terminating computation. In that proof (cf. Theorem 3.22) we embed an isolated computation that yields from one procedure call to another one, into a computation context. To be more precise, given a computation between two procedure calls:

$$\Gamma \vdash \langle [\mathit{the}\ (\Gamma\ p)],\ css,\ \mathit{Normal}\ s\rangle \to^{+} \langle [\mathit{the}\ (\Gamma\ q)],\ css',\ \mathit{Normal}\ t\rangle, \qquad (*)$$

we can embed this computation into another one, by appending continuations (cf. Lemma 3.20):

$$\Gamma \vdash \langle [\mathit{the}\ (\Gamma\ p)],\ css\ @\ css'',\ \mathit{Normal}\ s\rangle \to^{+} \langle [\mathit{the}\ (\Gamma\ q)],\ css'\ @\ css'',\ \mathit{Normal}\ t\rangle.$$

The *css''* is the computation rest that is accumulated before the procedure call to *p* is reached. This outer computation only affects the continuation stack because the CALL Rule cleans up the pending statements. Otherwise, with rule CALLSIMPLE, the outer computation would also affect the pending statements, since

$$\ldots \to \langle \mathit{Call}\ p{\cdot}cs, \ldots\rangle \to \langle \mathit{the}\ (\Gamma\ p){\cdot}cs, \ldots\rangle \to \ldots.$$

Due to the block structure of a computation in the small-step semantics we cannot just append some statements *cs* to the pending statements of (∗) to arrive at:

$$\Gamma \vdash \langle \mathit{the}\ (\Gamma\ p){\cdot}cs,\ css,\ \mathit{Normal}\ s\rangle \to^{+} \langle \mathit{the}\ (\Gamma\ q){\cdot}cs,\ css',\ \mathit{Normal}\ t\rangle.$$

Consider exiting a block:

$$\Gamma \vdash \langle [],\ (nrms,\ abrs){\cdot}css,\ \mathit{Normal}\ s\rangle \to \langle nrms,\ css,\ \mathit{Normal}\ s\rangle.$$

For a nonempty *cs* we do not get:

$$\Gamma \vdash \langle cs,\ (nrms,\ abrs){\cdot}css,\ \mathit{Normal}\ s\rangle \to \langle nrms\ @\ cs,\ css,\ \mathit{Normal}\ s\rangle.$$

Quite the opposite, first *cs* is executed instead of *nrms*.

With the small-step semantics we have all the preliminaries to define the termination ordering $\prec^{\Gamma}$ that we need for the CALLREC Rule. Remember that it is defined for pairs, consisting of a state and the procedure name. So $(t, q) \prec^{\Gamma} (s, p)$ means that procedure call *q* in state *t* comes after procedure call *p* in state *s*. Therefore it is "nearer" to the end of the terminating computation and thus regarded as smaller:

$$(\prec^{\Gamma}) \equiv \{((t,q),(s,p)).\ \Gamma \vdash \mathit{the}\ (\Gamma\ p) \downarrow \mathit{Normal}\ s\ \wedge \\ (\exists css.\ \Gamma \vdash \langle [\mathit{the}\ (\Gamma\ p)],\ [],\ \mathit{Normal}\ s\rangle \to^{+} \langle [\mathit{the}\ (\Gamma\ q)],\ css,\ \mathit{Normal}\ t\rangle)\}$$

◂ Definition 3.14

Conceptually $\prec^{\Gamma}$ relates two subsequent procedure call configurations. Since we use the transitive closure $\to^{+}$ intermediate procedure calls are not ruled out. Therefore the two procedure calls do not have to be strictly consecutive. To prove that $\prec^{\Gamma}$ is well-founded we string together subsequent calls of the form:

$$\Gamma\vdash \langle[\mathit{the}\ (\Gamma\ p)], [], \mathit{Normal}\ s\rangle \rightarrow^{+} \langle[\mathit{the}\ (\Gamma\ q)], \mathit{css}, \mathit{Normal}\ t\rangle.$$

We argue that this sequence cannot be infinite, since the initial configuration terminates. To string the isolated subsequent procedure calls together, the stack of continuations *css* is accumulated. For example:

1. $\Gamma\vdash \langle[\mathit{the}\ (\Gamma\ p_1)], [], \mathit{Normal}\ s_1\rangle \rightarrow^{+} \langle[\mathit{the}\ (\Gamma\ p_2)], \mathit{css}_2, \mathit{Normal}\ s_2\rangle$
2. $\Gamma\vdash \langle[\mathit{the}\ (\Gamma\ p_2)], [], \mathit{Normal}\ s_2\rangle \rightarrow^{+} \langle[\mathit{the}\ (\Gamma\ p_3)], \mathit{css}_3, \mathit{Normal}\ s_3\rangle$

We can continue the first computation by starting the second one with the continuation stack css_2:

1. $\Gamma\vdash \langle[\mathit{the}\ (\Gamma\ p_1)], [], \mathit{Normal}\ s_1\rangle \rightarrow^{+} \langle[\mathit{the}\ (\Gamma\ p_2)], \mathit{css}_2, \mathit{Normal}\ s_2\rangle$
2. $\Gamma\vdash \langle[\mathit{the}\ (\Gamma\ p_2)], \mathit{css}_2, \mathit{Normal}\ s_2\rangle \rightarrow^{+} \langle[\mathit{the}\ (\Gamma\ p_3)], \mathit{css}_3\ @\ \mathit{css}_2, \mathit{Normal}\ s_3\rangle$

Extending the continuation stack is justified by the following two lemmas:

Lemma 3.19 ► *If* $\Gamma\vdash \langle cs, css, s\rangle \rightarrow \langle cs', css', t\rangle$ *then* $\Gamma\vdash \langle cs, css\ @\ xs, s\rangle \rightarrow \langle cs', css'\ @\ xs, t\rangle$.

Proof. By induction on the single step relation. Note that there is no (recursive) appearance of the single step relation among the premises of the rules in Figure 3.4. Hence induction on the single step relation coincides with a case distinction on the rules. □

Induction on the transitive closure lifts this result to $\rightarrow^{+}$:

Lemma 3.20 ► *If* $\Gamma\vdash \langle cs, css, s\rangle \rightarrow^{+} \langle cs', css', t\rangle$ *then* $\Gamma\vdash \langle cs, css\ @\ xs, s\rangle \rightarrow^{+} \langle cs', css'\ @\ xs, t\rangle$.

In analogy to the infinite descending chain in Lemma 3.16 we introduce the notion of an infinite computation. $\Gamma\vdash \langle cs,css,s\rangle \rightarrow \ldots(\infty)$ expresses that there is an infinite computation starting in configuration $\langle cs, css, s\rangle$:

Definition 3.15 ►
Infinite computation

$$\Gamma\vdash \langle cs,css,s\rangle \rightarrow \ldots(\infty) \equiv \exists f.\, f\ 0 = \langle cs, css, s\rangle \wedge (\forall i.\ \Gamma\vdash f\ i \rightarrow f\ (i + 1))$$

Now we have two ways to describe termination of Simpl programs. By the termination judgement $\Gamma\vdash c \downarrow s$ and by the absence of an infinite computation. In fact both notions are equivalent:

Theorem 3.21 ►
Termination iff no infinite computation

$$\Gamma\vdash c \downarrow s = (\neg\ \Gamma\vdash \langle[c],[],s\rangle \rightarrow \ldots(\infty))$$

To keep the focus on completeness, the proof of this theorem is postponed to appendix A as Theorem A.20. It requires quite a lot of intermediate steps about the relation of the big- and the small-step semantics and about properties of infinite computations.

Theorem 3.22 ►

$$\mathit{wf}\ (<^{\Gamma})$$

Proof. We do the proof by contradiction. We assume that $<^{\Gamma}$ is not well-founded. According to Lemma 3.16 this means that there is an infinite descending chain for the relation $<^{\Gamma}$. From this assumptions we derive a contradiction. With $<^{\Gamma}$ we relate procedure call configurations, consisting of the procedure name and the state. This means that we assume that there is an infinite sequence of procedure names and program states each of which are terminating, and that are subsequently reachable from each other, formally:

$$\begin{aligned}&\forall i.\ \Gamma\vdash \mathit{the}\ (\Gamma\ (p\ i)) \downarrow \mathit{Normal}\ (s\ i)\ \wedge\\ &\quad(\exists \mathit{css}.\ \Gamma\vdash \langle[\mathit{the}\ (\Gamma\ (p\ i))], [], \mathit{Normal}\ (s\ i)\rangle \rightarrow^{+}\\ &\qquad\qquad \langle[\mathit{the}\ (\Gamma\ (p\ (i+1)))], \mathit{css}, \mathit{Normal}\ (s\ (i+1))\rangle).\end{aligned} \tag{$*$}$$

The function p enumerates the procedure names and s enumerates the states. From instantiating $(*)$ with 0 we get termination of the computation started in the initial configuration: $\Gamma\vdash \mathit{the}\ (\Gamma\ (p\ 0)) \downarrow \mathit{Normal}\ (s\ 0)$. With Theorem 3.21 (used from left to right) we therefore know that there is no infinite computation:

$$\nexists f.\ f\ 0 = \langle[\mathit{the}\ (\Gamma\ (p\ 0))], [], \mathit{Normal}\ (s\ 0)\rangle \wedge (\forall i.\ \Gamma\vdash f\ i \rightarrow f\ (i+1)). \tag{$**$}$$

The further strategy is to contradict this by constructing such an infinite computation from the isolated computation fragments between two subsequent procedure calls that we can obtain form $(*)$. Every such fragment starts with an empty continuation stack and accumulates a continuation stack *css* as it reaches the next procedure call. This continuation *css* describes the remaining computation that has to be executed after the second procedure call returns. To get the entire computation starting from the initial configuration out of those fragments, we have to accumulate the continuation stacks. To construct the infinite sequence of configurations the only piece missing is the sequence of continuation stacks. For each index i the current procedure can be obtained from the enumeration p and the current state from enumeration s. In $(*)$ the continuation stack *css* is existentially quantified under the universal quantification of i. Therefore it depends on i. To describe the sequence of continuation stacks we want to construct an enumeration function *css* in analogy to p and s. We use the axiom of choice to transform the $\forall i.\ \exists \mathit{css}.\ \ldots$ of $(*)$ into an enumeration function *css*. The axiom of choice reads as follows in HOL:

$$(\forall x.\ \exists y.\ Q\ x\ y) \longrightarrow (\exists f.\ \forall x.\ Q\ x\ (f\ x)).$$

If a predicate Q holds for an existential quantified y and an universally quantified outer x, then we can obtain a choice function f that selects the proper y from the x so that $Q\ x\ (f\ x)$ holds. This is exactly our situation. With the axiom of choice and $(*)$ we obtain the enumeration function *css* for the continuation stack of a fragment of computation:

$$\begin{aligned}&\forall i.\ \Gamma\vdash \langle[\mathit{the}\ (\Gamma\ (p\ i))], [], \mathit{Normal}\ (s\ i)\rangle \rightarrow^{+}\\ &\qquad \langle[\mathit{the}\ (\Gamma\ (p\ (i+1)))], \mathit{css}\ i, \mathit{Normal}\ (s\ (i+1))\rangle.\end{aligned} \tag{$*{*}*$}$$

By sequencing the *css* we can construct the complete computation from these fragments:

$$\begin{aligned}&\langle[\mathit{the}\ (\Gamma\ (p\ 0))], [], \mathit{Normal}\ (s\ 0)\rangle \rightarrow^{+}\\ &\langle[\mathit{the}\ (\Gamma\ (p\ 1))], \mathit{css}\ 0, \mathit{Normal}\ (s\ 1)\rangle \rightarrow^{+}\\ &\langle[\mathit{the}\ (\Gamma\ (p\ 2))], \mathit{css}\ 1\ @\ \mathit{css}\ 0, \mathit{Normal}\ (s\ 2)\rangle \rightarrow^{+} \ldots.\end{aligned}$$

The auxiliary function *seq* is used to sequence the *css*:

$$\begin{aligned}\mathit{seq}\ \mathit{css}\ 0 &= []\\ \mathit{seq}\ \mathit{css}\ (i+1) &= \mathit{css}\ i\ @\ \mathit{seq}\ \mathit{css}\ i\end{aligned}$$

We define an enumeration function f of configurations that form the infinite computation:

$$f \equiv \lambda i.\ ([\mathit{the}\ (\Gamma\ (p\ i))], \mathit{seq}\ \mathit{css}\ i, \mathit{Normal}\ (s\ i)).$$

Hence $f\ 0$ is our initial configuration:

$$f\ 0 = ([\mathit{the}\ (\Gamma\ (p\ 0))], [], \mathit{Normal}\ (s\ 0)).$$

Moreover, since a computation fragment of (∗∗∗):

$$\Gamma\vdash \langle[\mathit{the}\ (\Gamma\ (p\ i))], [], \mathit{Normal}\ (s\ i)\rangle \rightarrow^{+} \langle[\mathit{the}\ (\Gamma\ (p\ (i + 1)))], \mathit{css}\ i, \mathit{Normal}\ (s\ (i + 1))\rangle,$$

can be inserted into the complete computation by Lemma 3.19:

$$\begin{array}{l}\Gamma\vdash \langle[\mathit{the}\ (\Gamma\ (p\ i))], \mathit{seq\ css}\ i, \mathit{Normal}\ (s\ i)\rangle \rightarrow^{+}\\ \quad\langle[\mathit{the}\ (\Gamma\ (p\ (i + 1)))], \mathit{css}\ i\ @\ \mathit{seq\ css}\ i, \mathit{Normal}\ (s\ (i + 1))\rangle,\end{array}$$

we get an infinite computation $\forall i.\ \Gamma\vdash f\ i \rightarrow^{+} f\ (i + 1)$. This contradicts (∗∗). □

Deriving the MGT Equipped with the termination ordering $\prec^{\Gamma}_{b,c}$ for loops and $\prec^{\Gamma}$ for procedure calls we can continue the completeness proof along the lines of partial correctness.

Lemma 3.23 ► *MGT implies completeness*

Provided that the most general triple is derivable within the Hoare logic:

$$\begin{array}{l}\forall Z.\ \Gamma\vdash_{t/F} \{s.\ s = Z \wedge \Gamma\vdash \langle c,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\} \cup \mathit{Fault}\ `\ (-\ F) \wedge \Gamma\vdash c \downarrow \mathit{Normal}\ s\}\\ \qquad\qquad c\\ \qquad\qquad \{t.\ \Gamma\vdash \langle c,\mathit{Normal}\ Z\rangle \Rightarrow \mathit{Normal}\ t\},\{t.\ \Gamma\vdash \langle c,\mathit{Normal}\ Z\rangle \Rightarrow \mathit{Abrupt}\ t\},\end{array}$$

then every valid triple $\Gamma\models_{t/F} P\ c\ Q,A$ *is derivable in the Hoare logic:* $\Gamma\vdash_{t/F} P\ c\ Q,A$.

Proof. The proof is analogous to the proof of Lemma 3.9. □

The next step is to derive the MGT under the assumption that the MGT of all procedures is derivable.

Lemma 3.24 ► *Provided that the MGT for all procedures in* Γ *is derivable:*

$$\begin{array}{l}\forall p\in \mathit{dom}\ \Gamma.\\ \quad\forall Z.\ \Gamma,\Theta\vdash_{t/F}\\ \qquad\quad \{s.\ s = Z \wedge \Gamma\vdash \langle \mathit{Call}\ p,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\} \cup \mathit{Fault}\ `\ (-\ F) \wedge \Gamma\vdash \mathit{Call}\ p \downarrow \mathit{Normal}\ s\}\\ \qquad\quad \mathit{Call}\ p\\ \qquad\quad \{t.\ \Gamma\vdash \langle \mathit{Call}\ p,\mathit{Normal}\ Z\rangle \Rightarrow \mathit{Normal}\ t\},\{t.\ \Gamma\vdash \langle \mathit{Call}\ p,\mathit{Normal}\ Z\rangle \Rightarrow \mathit{Abrupt}\ t\},\end{array}$$

then the MGT for command c is also derivable:

$$\begin{array}{l}\forall Z.\ \Gamma,\Theta\vdash_{t/F}\\ \qquad \{s.\ s = Z \wedge \Gamma\vdash \langle c,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\} \cup \mathit{Fault}\ `\ (-\ F) \wedge \Gamma\vdash c \downarrow \mathit{Normal}\ s\}\\ \qquad c\\ \qquad \{t.\ \Gamma\vdash \langle c,\mathit{Normal}\ Z\rangle \Rightarrow \mathit{Normal}\ t\},\{t.\ \Gamma\vdash \langle c,\mathit{Normal}\ Z\rangle \Rightarrow \mathit{Abrupt}\ t\}.\end{array}$$

Proof. By induction on the syntax of command c and along the lines of the proof of Lemma 3.10. For compound statements the termination restriction $\Gamma\vdash c \downarrow \mathit{Normal}\ s$ in the precondition is decomposed in the same fashion as the exclusion of stuck and faulty computations $\Gamma\vdash \langle c,\mathit{Normal}\ s\rangle \Rightarrow\notin\{\mathit{Stuck}\} \cup \mathit{Fault}\ `\ (-\ F)$ in Lemma 3.10.

In case of *While b c* the invariant of Lemma 3.10 is strengthened with the fact that from every intermediate state t that is reachable by unrolling the loop, execution of *While b c* also terminates:

- $\mathit{unroll} \equiv \{(s, t).\ s \in b \wedge \Gamma\vdash \langle c,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Normal}\ t\}^{*}$

- $P' \equiv$
  $\lambda Z.\ \{t.\ (Z, t) \in \mathit{unroll} \wedge$
  $\quad (\forall s_1.\ (Z, s_1) \in \mathit{unroll} \longrightarrow$
  $\quad\quad s_1 \in b \longrightarrow$
  $\quad\quad \Gamma\vdash \langle c, \mathit{Normal}\ s_1\rangle \Rightarrow\notin \{\mathit{Stuck}\} \cup \mathit{Fault}\ `\ (-\ F) \wedge$
  $\quad\quad (\forall s_2.\ \Gamma\vdash \langle c, \mathit{Normal}\ s_1\rangle \Rightarrow \mathit{Abrupt}\ s_2 \longrightarrow$
  $\quad\quad\quad \Gamma\vdash \langle \mathit{While}\ b\ c, \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Abrupt}\ s_2)) \wedge$
  $\quad \Gamma\vdash \mathit{While}\ b\ c \downarrow \mathit{Normal}\ t\}$

The well-founded relation required by the WHILE Rule is of course the termination ordering $\prec^{\Gamma}_{b,c}$ for loops. □

To discharge the precondition of the previous lemma we have to derive the MGT for all procedures p in Γ. With the following lemma we argue that for all statements c that are reachable from procedure p in initial state σ, it is sufficient to assume that the MGT of all smaller procedure calls with respect to $\prec^{\Gamma}$ are derivable. These are exactly those MGTs that are made available as assumptions in the CALLREC Rule.

◂ Lemma 3.25

Provided that the MGT for all procedure configurations (s, q) *that are smaller than the initial configuration* (σ, p) *with respect to* $\prec^{\Gamma}$ *are derivable:*

$$
\begin{array}{l}
\forall q{\in}\mathit{dom}\ \Gamma.\\
\quad \forall Z.\ \Gamma,\Theta\vdash_{t/F}\\
\quad\quad \{s.\ s = Z \wedge\\
\quad\quad\quad \Gamma\vdash \langle \mathit{Call}\ q, \mathit{Normal}\ s\rangle \Rightarrow\notin \{\mathit{Stuck}\} \cup \mathit{Fault}\ `\ (-\ F) \wedge\\
\quad\quad\quad \Gamma\vdash \mathit{Call}\ q \downarrow \mathit{Normal}\ s \wedge (s, q) \prec^{\Gamma} (\sigma, p)\}\\
\quad\quad \mathit{Call}\ q\\
\quad\quad \{t.\ \Gamma\vdash \langle \mathit{Call}\ q, \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Normal}\ t\},\{t.\ \Gamma\vdash \langle \mathit{Call}\ q, \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Abrupt}\ t\},
\end{array}
$$

then the MGT for all statements c *that are reachable from the initial configuration are derivable:*

$$
\begin{array}{l}
\forall Z.\ \Gamma,\Theta\vdash_{t/F}\\
\quad \{s.\ s = Z \wedge\\
\quad\quad \Gamma\vdash \langle c, \mathit{Normal}\ s\rangle \Rightarrow\notin \{\mathit{Stuck}\} \cup \mathit{Fault}\ `\ (-\ F) \wedge\\
\quad\quad \Gamma\vdash \mathit{the}\ (\Gamma\ p) \downarrow \mathit{Normal}\ \sigma \wedge\\
\quad\quad (\exists cs\ css.\ \Gamma\vdash \langle [\mathit{the}\ (\Gamma\ p)], [], \mathit{Normal}\ \sigma\rangle \rightarrow^{*} \langle c{\cdot}cs, css, \mathit{Normal}\ s\rangle)\}\\
\quad c\\
\quad \{t.\ \Gamma\vdash \langle c, \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Normal}\ t\},\{t.\ \Gamma\vdash \langle c, \mathit{Normal}\ Z\rangle \Rightarrow \mathit{Abrupt}\ t\}.
\end{array}
$$

Proof. By induction on the syntax of command c and along the lines of the proof of Lemma 3.10 or Lemma 3.24. From the precondition of the Hoare triple we know that the current configuration is reachable from the initial state:

$$\Gamma\vdash \langle [\mathit{the}\ (\Gamma\ p)], [], \mathit{Normal}\ \sigma\rangle \rightarrow^{*} \langle c{\cdot}cs, css, \mathit{Normal}\ s\rangle.$$

To get hold of the induction hypothesis for compound statements this computation has to be extended until the sub-statement is the head of the statement list. For example, consider the case of sequential composition $c_1; c_2$. In this case we know from the precondition:

$$\Gamma\vdash \langle [\mathit{the}\ (\Gamma\ p)], [], \mathit{Normal}\ \sigma\rangle \rightarrow^{*} \langle (c_1; c_2){\cdot}cs, css, \mathit{Normal}\ s\rangle.$$

According to the small-step semantics *Seq* is decomposed and the next configuration is $\langle c_1 \cdot c_2 \cdot cs,\ css,\ Normal\ s\rangle$ and therefore we can make use of the induction hypothesis for c_1, since statement c_1 is the head of the statement list. From the postcondition for normal termination of the MGT for c_1 we arrive at a final state t according to the big-step semantics:

$$\Gamma \vdash \langle c_1, Normal\ s\rangle \Rightarrow Normal\ t.$$

With Lemma A.1 we can extend the small-step computation to the configuration $\langle c_2 \cdot cs,\ css,\ Normal\ t\rangle$ and thus make use of the MGT for c_2.

For abrupt termination of c_1 the argument is analogous.

Case *Call q*. From the precondition of the Hoare Triple we know that the computation does not get stuck and thus $q \in dom\ \Gamma$. Moreover, the current configuration is reachable form the initial one:

$$\Gamma \vdash \langle [the\ (\Gamma\ p)],\ [],\ Normal\ \sigma\rangle \rightarrow^{*} \langle Call\ q \cdot cs,\ css,\ Normal\ s\rangle.$$

The next configuration according to the small-step semantics is

$$\langle [the\ (\Gamma\ q)], (cs, Throw \cdot cs) \cdot css, Normal\ s\rangle.$$

Hence we have $(s, q) \prec^{\Gamma} (\sigma, p)$ and can use the consequence rule to adapt the MGT for all reachable procedure calls from the assumption of the lemma.

Case *While b c*: The invariant of Lemma 3.24 is strengthened with the fact that from every intermediate state t that is reached by unrolling the loop, *While b c* in this state t is reachable from the initial configuration:

- $unroll \equiv \{(s, t).\ s \in b \wedge \Gamma \vdash \langle c, Normal\ s\rangle \Rightarrow Normal\ t\}^{*}$
- $P' \equiv$

```
$\lambda Z. \{t. (Z, t) \in unroll\ \wedge$
        $(\forall s_1. (Z, s_1) \in unroll \longrightarrow$
             $s_1 \in b \longrightarrow$
             $\Gamma \vdash \langle c, Normal\ s_1\rangle \Rightarrow \notin \{Stuck\} \cup Fault\ `\ (-\ F)\ \wedge$
             $(\forall s_2. \Gamma \vdash \langle c, Normal\ s_1\rangle \Rightarrow Abrupt\ s_2 \longrightarrow$
                    $\Gamma \vdash \langle While\ b\ c, Normal\ Z\rangle \Rightarrow Abrupt\ s_2))\ \wedge$
        $\Gamma \vdash the\ (\Gamma\ p) \downarrow Normal\ \sigma\ \wedge$
        $(\exists cs\ css. \Gamma \vdash \langle [the\ (\Gamma\ p)], [], Normal\ \sigma\rangle \rightarrow^{*} \langle While\ b\ c \cdot cs, css, Normal\ t\rangle)\}$
```

The well-founded relation required by the WHILE Rule is of course the termination ordering $\prec^{\Gamma}_{b,c}$ for loops. □

We instantiate statement c of the previous lemma with the body of the initial procedure *the* $(\Gamma\ p)$ and fix the initial state σ. Since the initial state is trivially reachable, the consequence rule allows us to conclude that the MGT of the initial procedure body is reachable from the restricted MGTs for the procedure calls:

Lemma 3.26 ► *Provided that the MGT for all procedure configurations (s, q) that are smaller than the initial configuration (σ, p) with respect to $\prec^{\Gamma}$ are derivable:*

```
$\forall q \in dom\ \Gamma.$
  $\forall Z. \Gamma, \Theta \vdash_{t/F}$
        $\{s. s = Z\ \wedge$
           $\Gamma \vdash \langle Call\ q, Normal\ s\rangle \Rightarrow \notin \{Stuck\} \cup Fault\ `\ (-\ F)\ \wedge$
           $\Gamma \vdash Call\ q \downarrow Normal\ s \wedge (s, q) \prec^{\Gamma} (\sigma, p)\}$
```

$$\begin{array}{l}
\quad \textit{Call } q\\
\quad \{t.\ \Gamma\vdash \langle \textit{Call } q, \textit{Normal } Z\rangle \Rightarrow \textit{Normal } t\},\{t.\ \Gamma\vdash \langle \textit{Call } q, \textit{Normal } Z\rangle \Rightarrow \textit{Abrupt } t\},
\end{array}$$

then the MGT of the initial procedure body is derivable:

$$\begin{array}{l}
\forall Z.\ \Gamma,\Theta\vdash_{t/F}\\
\quad (\{\sigma\} \cap\\
\quad\ \{s.\ s = Z \land\\
\quad\quad \Gamma\vdash \langle \textit{the } (\Gamma\ p), \textit{Normal } s\rangle \Rightarrow\notin \{\textit{Stuck}\} \cup \textit{Fault}\ `\ (-\ F) \land \Gamma\vdash \textit{the } (\Gamma\ p) \downarrow \textit{Normal } s\})\\
\quad \textit{the } (\Gamma\ p)\\
\quad \{t.\ \Gamma\vdash \langle \textit{the } (\Gamma\ p), \textit{Normal } Z\rangle \Rightarrow \textit{Normal } t\},\{t.\ \Gamma\vdash \langle \textit{the } (\Gamma\ p), \textit{Normal } Z\rangle \Rightarrow \textit{Abrupt } t\}
\end{array}$$

Proof. This lemma is an instance of Lemma 3.25. □

Now we are in a situation where we can apply the CallRec Rule (cf. p. 56) and can conclude that the MGT of all procedure calls is derivable:

◀ Lemma 3.27

$$\begin{array}{l}
\forall p\in \textit{dom } \Gamma.\\
\ \forall Z.\ \Gamma,\Theta\vdash_{t/F}\\
\quad \{s.\ s = Z \land\\
\quad\quad \Gamma\vdash \langle \textit{Call } p, \textit{Normal } s\rangle \Rightarrow\notin \{\textit{Stuck}\} \cup \textit{Fault}\ `\ (-\ F) \land \Gamma\vdash \textit{Call } p \downarrow \textit{Normal } s\}\\
\quad \textit{Call } p\\
\quad \{t.\ \Gamma\vdash \langle \textit{Call } p, \textit{Normal } Z\rangle \Rightarrow \textit{Normal } t\},\{t.\ \Gamma\vdash \langle \textit{Call } p, \textit{Normal } Z\rangle \Rightarrow \textit{Abrupt } t\}
\end{array}$$

Proof. We attempt to use the CallRec Rule with $\prec^\Gamma$ as well-founded relation. We define the set of specifications *Specs* as the MGT for all procedure calls to defined procedures:

$$\begin{array}{l}
\textit{Specs} =\\
(\bigcup_{p\in \textit{dom } \Gamma}\\
\quad \bigcup_Z \{(\{s.\ s = Z \land\\
\quad\quad\quad \Gamma\vdash \langle \textit{Call } p, \textit{Normal } s\rangle \Rightarrow\notin \{\textit{Stuck}\} \cup \textit{Fault}\ `\ (-\ F) \land \Gamma\vdash \textit{Call } p \downarrow \textit{Normal } s\},\\
\quad\quad p,\\
\quad\quad \{t.\ \Gamma\vdash \langle \textit{Call } p, \textit{Normal } Z\rangle \Rightarrow \textit{Normal } t\},\\
\quad\quad \{t.\ \Gamma\vdash \langle \textit{Call } p, \textit{Normal } Z\rangle \Rightarrow \textit{Abrupt } t\})\})
\end{array}$$

According to the premise of the CallRec Rule we define the restriction of the specifications for smaller configurations with respect to $\prec^\Gamma$:

$$\textit{Specs-wf } p\ \sigma = (\lambda(P, q, Q, A).\ (P \cap \{s.\ (s, q) \prec^\Gamma (\sigma, p)\}, q, Q, A))\ `\ \textit{Specs}.$$

For any procedure $p \in \textit{dom } \Gamma$ and any initial state σ, within context *Specs-wf p σ* we can use Lemma 3.26 to derive the MGT for the procedure body of p. Lemma 3.26 is applicable since its premise can be solved by the assumption rule Asm, because we are in context *Specs-wf p σ*. Hence we have:

$$\begin{array}{l}
\Gamma,\textit{Specs-wf } p\ \sigma\vdash_{t/F}\\
\quad (\{\sigma\} \cap\\
\quad\ \{s.\ s = Z \land\\
\quad\quad \Gamma\vdash \langle \textit{the } (\Gamma\ p), \textit{Normal } s\rangle \Rightarrow\notin \{\textit{Stuck}\} \cup \textit{Fault}\ `\ (-\ F) \land \Gamma\vdash \textit{the } (\Gamma\ p) \downarrow \textit{Normal } s\})\\
\quad \textit{the } (\Gamma\ p)\\
\quad \{t.\ \Gamma\vdash \langle \textit{the } (\Gamma\ p), \textit{Normal } Z\rangle \Rightarrow \textit{Normal } t\},\{t.\ \Gamma\vdash \langle \textit{the } (\Gamma\ p), \textit{Normal } Z\rangle \Rightarrow \textit{Abrupt } t\}
\end{array}$$

This is almost what we need to prove about the procedure bodies according to the CallRec Rule. We only have to replace the occurences of *the* $(\Gamma\ p)$ in the pre- and postcondition by *Call p*. Since procedure p is defined we have the following equivalences:

- $\Gamma\vdash \langle Call\ p,s\rangle \Rightarrow t = \Gamma\vdash \langle the\ (\Gamma\ p),s\rangle \Rightarrow t$
- $\Gamma\vdash \langle Call\ p,Normal\ s\rangle \Rightarrow\notin\{Stuck\} \cup Fault\ `\ (-\ F) =$
  $\Gamma\vdash \langle the\ (\Gamma\ p),Normal\ s\rangle \Rightarrow\notin\{Stuck\} \cup Fault\ `\ (-\ F)$
- $\Gamma\vdash Call\ p \downarrow Normal\ s = \Gamma\vdash the\ (\Gamma\ p) \downarrow Normal\ s.$

With these equivalences we can refine the Hoare triple to:

$$\begin{array}{l}\Gamma,Specs\text{-}wf\ p\ \sigma\vdash_{t/F}\\ \quad(\{\sigma\} \cap\\ \quad\{s.\ s = Z \wedge \Gamma\vdash \langle Call\ p,Normal\ s\rangle \Rightarrow\notin\{Stuck\} \cup Fault\ `\ (-\ F) \wedge \Gamma\vdash Call\ p \downarrow Normal\ s\})\\ \quad the\ (\Gamma\ p)\\ \quad\{t.\ \Gamma\vdash \langle Call\ p,Normal\ Z\rangle \Rightarrow Normal\ t\},\{t.\ \Gamma\vdash \langle Call\ p,Normal\ Z\rangle \Rightarrow Abrupt\ t\}\end{array}$$

This is the required Hoare triple for the CallRec Rule. □

This lemma discharges the assumption of Lemma 3.24. Therefore we have proven that the MGT is derivable in the Hoare logic. With Lemma 3.23 we arrive at the completeness theorem for total correctness:

Theorem 3.28 ► *Completeness*

$$\Gamma\models_{t/F} P\ c\ Q,A \longrightarrow \Gamma\vdash_{t/F} P\ c\ Q,A$$

To obtain completeness within context Θ we again have to augment the Hoare logic with an additional rule that allows to derive invalid triples if there is a wrong specification among the assumptions:

$$\frac{\Gamma,\Theta\models_{t/F} P\ c\ Q,A \qquad \neg\ \Gamma\models_{t/F} P\ c\ Q,A}{\Gamma,\Theta\vdash_{t/F} P\ c\ Q,A}\ (\text{ExFalso})$$

Lemma 3.29 ► *Completeness within context*

If we augment the Hoare logic with the ExFalso *Rule then:*

$$\Gamma,\Theta\models_{t/F} P\ c\ Q,A \longrightarrow \Gamma,\Theta\vdash_{t/F} P\ c\ Q,A$$

3.3 Conclusion

In this chapter I have introduced a Hoare logic for partial and total correctness of Simpl programs and have presented the soundness and completeness proofs. The extensional representation of assertions as HOL sets makes the definition of an assertion logic unnecessary. Besides, the flexibility to use arbitrary HOL predicates as assertions also simplifies the completeness proof, since we can directly use the operational semantics in the assertions without having to deal with expressivity issues or an encoding of the semantics in the assertion logic. Albeit the expressive power of Simpl, the high level of abstraction leads to concise soundness and completeness proofs. Especially the completeness proof for total correctness goes further than related work in the area of formalised and machine checked program calculi. This proof extends the work of Nipkow [77] to Simpl, which can in particular handle unbounded nondeterminism, abrupt termination, dynamic method invocation and higher order features like pointers to procedurs and closures.

The handling of auxiliary variables and the consequence rule are further clarified. It turns out that there is no need to mention auxiliary variables at all in the core calculus.

The Hoare logic provides extended means to reason about runtime faults. Chapter 5 explains how this feature provides an interface to integrate program analysis result into the Hoare logic.

CHAPTER **4**

Utilising the Hoare Logic

This chapter describes the integration and automation of the Hoare logic in Isabelle. A verification condition generator is built as Isabelle tactic on top of the Hoare logic. Examples illustrate how we deal with various aspects of the programming language and what the resulting proof obligations look like.

Contents

In this chapter, we introduce Hoare logic rules for the derived statements like the procedure call with parameters. These rules can either be derived directly from the basic set of Hoare logic rules, or we can prove their validity and use the completeness Theorems 3.12 and 3.28 to introduce them. In neither case we have to augment the inductive definitions of the Hoare rules.

Our main tool is a verification condition generator that is implemented as tactic called *vcg*. The Hoare logic rules in Figures 3.1 and 3.3 are syntax directed and defined in a weakest precondition style, so that we can almost take them as they are. With the consequence rule, we derive variants of the Hoare rules where all assertions in the conclusions are plain variables so that they are applicable to every context. We get the following format:

$$\frac{P \subseteq WP \ldots}{\Gamma{,}\Theta \vdash P\ c\ Q{,}A}$$

The ... may be recursive Hoare triples or side-conditions which somehow lead to the weakest precondition *WP*. If we recursively apply rules of this format until the program c is completely processed, then we have calculated the weakest precondition WP and are left with the verification condition $P \subseteq WP$. The set inclusion is then transformed to an implication. Finally we split the state records so that the record representation does not show up in the resulting verification condition. This leads to quite comprehensible proof obligations that closely resemble the specifications.

Although the Hoare rules manipulate the state in the assertions, instead applying a syntactic substitutions, stepping through verification condition generation "feels" like the expected syntactic substitutions of traditional Hoare logic. As already described in the beginning of Section 3 this is achieved by simplification of the record updates in the assertions calculated by the Hoare rules. Here is a first example:

lemma
$\Gamma\vdash \{\!|\mathsf{m} = a \wedge \mathsf{n} = b|\!\}\ \mathsf{i} := \mathsf{m};\ \mathsf{m} := \mathsf{n};\ \mathsf{n} := \mathsf{i}\ \{\!|\mathsf{m} = b \wedge \mathsf{n} = a|\!\}$
apply *vcg-step*

1. $\Gamma\vdash \{\!|\mathsf{m} = a \wedge \mathsf{n} = b|\!\}\ \mathsf{i} := \mathsf{m};\ \mathsf{m} := \mathsf{n}\ \{\!|\mathsf{m} = b \wedge \mathsf{i} = a|\!\}$

apply *vcg-step*

1. $\Gamma\vdash \{\!|\mathsf{m} = a \wedge \mathsf{n} = b|\!\}\ \mathsf{i} := \mathsf{m}\ \{\!|\mathsf{n} = b \wedge \mathsf{i} = a|\!\}$

apply *vcg-step*

1. $\{\!|\mathsf{m} = a \wedge \mathsf{n} = b|\!\} \subseteq \{\!|\mathsf{n} = b \wedge \mathsf{m} = a|\!\}$

apply *vcg-step*

1. $\bigwedge m\ n.\ n = n \wedge m = m$

In the first three steps the sequential composition is processed by the Rules Seq and Basic and the postcondition is simplified to perform the substitution. In the last step the set inclusion is transformed to the corresponding implication and the state record is split. If we omit the state split we obtain the following verification condition:

$$\bigwedge s.\ n\ s = n\ s \wedge m\ s = m\ s$$

Symbol $\bigwedge$ is the universal quantifier of Isabelle's meta logic.

4.1 Loops

To verify a loop, the user annotates an invariant. For total correctness the user also supplies the variant, which in our case is a well-founded relation on the state space, which decreases by execution of the loop body. We introduce constant *whileAnno* to extend statement *While* with the annotations I for the invariant and V for the variant.

Definition 4.1 ►

$$\begin{array}{l} \mathit{whileAnno} :: {'s}\ \mathit{set} \Rightarrow {'s}\ \mathit{set} \Rightarrow ({'s} \times {'s})\ \mathit{set} \Rightarrow ({'s},{'p},{'f})\ \mathit{com} \Rightarrow ({'s},{'p},{'f})\ \mathit{com} \\ \mathit{whileAnno}\ b\ I\ V\ c \equiv \mathit{While}\ b\ c \end{array}$$

This definition reflects that the annotations are mere comments for the verification condition generator. The annotations do not appear on the right hand side. Rewriting a program with this definition yields a pure Simpl program without any annotations. For the annotated loop we derive the following rule for partial correctness that is used by the verification condition generator:

$$\frac{P \subseteq I \qquad \Gamma,\Theta\vdash_{/F} (I \cap b)\ c\ I,A \qquad I \cap -b \subseteq Q}{\Gamma,\Theta\vdash_{/F} P\ (whileAnno\ b\ I\ V\ c)\ Q,A}$$

The precondition has to imply the invariant, the loop body has to preserve the invariant and finally the invariant together with the negated loop condition has to imply the postcondition. The rule for total correctness also takes the variant annotation into account.

$$\frac{P \subseteq I \qquad \forall \sigma.\ \Gamma,\Theta\vdash_{t/F} (\{\sigma\} \cap I \cap b)\ c\ (\{t.\ (t, \sigma) \in V\} \cap I),A \qquad I \cap -b \subseteq Q \qquad wf\ V}{\Gamma,\Theta\vdash_{t/F} P\ (whileAnno\ b\ I\ V\ c)\ Q,A}$$

The state before execution of the loop body is fixed with σ. The state after execution of the loop body has to be smaller than σ with respect to the variant V.

Proving that a relation is well-founded can be quite hard. Fortunately there are ways to compose relations so that they are well-founded by construction. This infrastructure is already present in Isabelle/HOL [80], since the **recdef** command for general recursive functions builds on it. For example, *measure f* is always well-founded, where f is a function that maps the state to a natural number; the lexicographic product of two well-founded relations is again well-founded and the inverse image construction *inv-image* of a well-founded relation is again well-founded. These constructions are best explained by the following equations:

$$\begin{array}{lcl} (x, y) \in measure\ f & = & f\ x < f\ y \\ ((a, b), (x, y)) \in r <\!*lex*\!> s & = & (a, x) \in r \lor a = x \land (b, y) \in s \\ (x, y) \in inv\text{-}image\ r\ f & = & (f\ x, f\ y) \in r \end{array}$$

Another useful construction is $<\!*mlex*\!>$ which is a combination of a measure and a lexicographic product:

$$((x, y) \in f <\!*mlex*\!> r) = (f\ x < f\ y \lor f\ x = f\ y \land (x, y) \in r)$$

In contrast to the lexicographic product it does not construct a product type. The state may either decrease according to the measure function f or the measure stays the same and the state decreases accordint to relation r.

The following example calculates multiplication by iterated addition. The distance of the loop variable m to a decreases in every iteration. This is expressed by the measure function $a - \mathsf{m}$ on the state space.

lemma $\Gamma\vdash_t$ ⦃m = 0 ∧ s = 0⦄
WHILE m ≠ a **INV**⦃s = m ∗ b ∧ m ≤ a⦄ **VAR** *MEASURE* a − m
DO s := s + b; m := m + 1 **OD**
⦃s = a ∗ b⦄
apply *vcg*

1. $\bigwedge m\ s.\ [\![m = 0; s = 0]\!] \Longrightarrow s = m * b \land m \le a$
2. $\bigwedge m\ s.\ [\![s = m * b; m \le a; m \ne a]\!]$
   $\Longrightarrow a - (m + 1) < a - m \land s + b = (m + 1) * b \land m + 1 \le a$
3. $\bigwedge m\ s.\ [\![s = m * b; m \le a; \neg\ m \ne a]\!] \Longrightarrow s = a * b$

The invariant annotation is preceded by **INV** and variant annotation by **VAR**. The capital *MEASURE* is a shorthand for *measure* ($\lambda s.\ a - m\ s$). The three subgoals stem from the first three preconditions of the rule above. The well-foundedness of the variant was already discharged by the verification condition generator. The first one is for the path from the precondition to the invariant. The second one is for the loop body. The loop body has to decrease the variant and reestablish the invariant. The final subgoal guarantees that the invariant together with the negated loop condition implies the postcondition.

4.2 Expressions with Side Effects

In Section 2.4.3 we introduced the command *bind* to model expressions with side effects. The Hoare rule for *bind* is the following:

$$\frac{P \subseteq \{s.\ s \in P'\ s\} \qquad \forall s.\ \Gamma,\Theta \vdash_{/F} (P'\ s)\ (c\ (e\ s))\ Q,A}{\Gamma,\Theta \vdash_{/F} P\ (bind\ e\ c)\ Q,A}$$

The intuitive reading of the rule is backwards in the style of the weakest precondition calculation. The initial state is s. The postcondition we want to reach is Q or A. Since statement c depends on the initial state s via expression e, the intermediate assertion P' depends on s, too. Since it has to work for any initial state s it is universally quantified. The actual precondition P describes a subset of the states of the weakest precondition $P'\ s$. The rule for total correctness is structurally equivalent.

The following example is for the statement `r = m++ + n`:

lemma $\Gamma\vdash \{\!|True|\!\}\ \mathsf{m} \gg m.\ \mathsf{m} := \mathsf{m} + 1;\ \mathsf{r} := m + \mathsf{n}\ \{\!|\mathsf{r} = \mathsf{m} + \mathsf{n} - 1|\!\}$
apply *vcg*

1. $\bigwedge m\ n.\ True \Longrightarrow m + n = m + 1 + n - 1$

The initial values of the variables are m and n. So in the postcondition r is substituted by $m + n$ and m by $m + 1$.

4.3 Blocks

Local variables can be introduced with the *block* statement (cf. Section 2.4.5). For example, {`int i; i = 2`} is modelled by:

$$block\ (\lambda s.\ s)\ (\mathsf{i} := 2)\ (\lambda s\ t.\ t(\!|i := i\ s|\!))\ (\lambda s\ t.\ \textbf{SKIP}).$$

The initialisation function is the identity and the body of the block is the assignment. To exit the block, the initial value of i is restored. The follow-up statement is **SKIP**. A block introduces a dependency of the assertions on the initial state s and the final state of the body t. This is reflected in the Hoare rule for *block*. Again the intermediate states are universally quantified.

$$\frac{\begin{array}{c}P \subseteq \{s.\ init\ s \in P'\ s\} \qquad \forall s.\ \Gamma,\Theta\vdash_{/F} (P'\ s)\ bdy\ \{t.\ return\ s\ t \in R\ s\ t\},\{t.\ return\ s\ t \in A\}\\ \forall s\ t.\ \Gamma,\Theta\vdash_{/F} (R\ s\ t)\ (c\ s\ t)\ Q,A\end{array}}{\Gamma,\Theta\vdash_{/F} P\ (block\ init\ bdy\ return\ c)\ Q,A}$$

Figure 4.1 illustrates the connection between the Hoare rule and the execution of a *block* for normal termination.

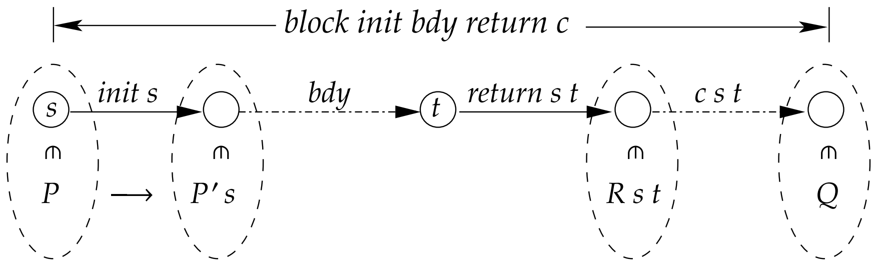

Figure 4.1: Illustration of the Hoare rule for a *block*

The operational reading of this rule is backwards, like it is applied by the verification condition generator. We start with the target postconditions Q and A. The follow-up statement c and the function *return* then introduces the dependency of s and t, which is propagated to the intermediate assertions R and P'. In the example above the return function resets the value of i to the initial value. Hence every occurrence of i in the postcondition refers to the initial value. We provide the concrete syntax *LOC* … *COL* to introduce locally bound variables. Next we show a property for the example:

lemma $\Gamma \vdash \{\!|\mathsf{i} = i|\!\}$ *LOC* i; i := *2* *COL* $\{\!|\mathsf{i} \le i|\!\}$
apply *vcg*

1. $\bigwedge i.\ i \le i$

As expected the value of i in the pre- and poststate coincides.

4.4 Procedures

To introduce a new procedure we provide the command **procedures**. Isabelle is programmable and offers an extensible top-level. The Hoare logic module uses this facility to introduce the new command.

procedures *Fac* (*n*|*r*) =
IF n = *0* **THEN** r := *1*
ELSE r := **CALL** *Fac*(n − *1*); r := n ∗ r **FI**

Fac-spec: $\forall n.\ \Gamma \vdash \{\!|\mathsf{n} = n|\!\}$ r := **PROC** *Fac*(n) $\{\!|\mathsf{r} = \mathit{fac}\ n|\!\}$

A procedure is given by its signature followed by its body and optionally some named specifications. The parameters in front of the pipe | are value parameters and behind the pipe are result parameters. Value parameters model call by value semantics. The value of a result parameter at the end of the procedure is passed back to the caller. Most common programming languages do not have the concept of a result parameter. However, our language is a model for sequential programs rather than a "real" programming language. We represent `return e` as an assignment of `e` to the result variable. In order to capture the abrupt termination stemming from a `return` we can use the techniques described in Section 2.4.4.

The command **procedures** automatically installs syntax translations for the procedure call. As running example we explain the procedure call m := **CALL** *Fac*(i). As introduced in Section 2.4.5 this translates to the internal form *call init "Fac" return result* with the proper *init*, *return* and *result* functions. Starting in an initial state *s* first the *init* function is applied, in order to pass the parameters. Then we execute the procedure body according to the environment Γ. Upon normal termination of the body in a state *t*, we first exit the procedure according to the function *return s t* and then continue execution with *result s t*. In case of an abrupt termination the final state is given by *return s t*. The function *return* passes back the global variables (and heap components) and restores the local variables of the caller, and *result* additionally assigns results to the scope of the caller. The *return*/*result* functions get both the initial state *s* before the procedure call and the final state *t* after execution of the body. If the body terminates abruptly we only apply the *return* function, thus the global state is propagated to the caller but no result is assigned. This is the expected semantics of an exception. For our example m := **CALL** *Fac*(i) the *init* function copies the actual parameter i to the formal parameter n, hence we have: $init\ s = s(\!|n := i\ s|\!)$. The *return* function updates the global variables of the initial state with their values in the final state. The global variables are all grouped together in a single record field: $return\ s\ t = s(\!|globals := globals\ t|\!)$. The *result* function is not just a state update function like *return*, but yields a complete command, like the second argument in the *bind* command. To model nested procedure calls we can use the same technique as described for side-effecting expressions. In our example the *result* statement is an assignment that copies the formal result parameter r to m: $result\ s\ t = Basic\ (\lambda u.\ u(\!|m := r\ t|\!))$. Here *s* is the initial state (before parameter passing), *t* the final state of the procedure body, and *u* the state after the *return* from the procedure. If the procedure has multiple result parameters this leads to a sequence of state-updates in the *result* statement.

Procedure specifications are ordinary Hoare triples. We use the parameterless call for the specification: r := **PROC** *Fac*(n) is syntactic sugar for *Call "Fac"*. This emphasises that the specification describes the internal behaviour of the procedure, whereas parameter passing corresponds to the procedure call. The precondition of the factorial specification fixes the current value n to the logical variable *n*. Universal quantification over *n* enables us to adapt the specification to an actual parameter. Besides providing convenient syntax, the command **procedures** also defines a constant for the procedure body (named *Fac-body*) and creates two so called *locales*. The purpose of locales is to set up logical contexts to support modular reasoning [10]. One locale is named like the specification, in our case *Fac-spec*. This locale contains the procedure specification. The second locale is named *Fac-impl* and contains the assumption $\Gamma\ "Fac" = \lfloor Fac\text{-}body \rfloor$, which expresses that the procedure is defined in the current environment. The purpose of these locales is to give us easy means to setup the context in which we prove programs correct. For example, we are not fixed to verify procedures in a strict bottom-up fashion. With the locales we can first only assume the specifications of auxiliary procedures and prove them later on. Hence a mixed bottom-up and top-down verification is possible.

Procedure Call By including Locale *Fac-spec*, the following lemma assumes that the specification of the factorial holds. If the specification is already proven it does not have to be explicitly included. The *vcg* uses the specification to handle the procedure call. The example also illustrates locality of i.

lemma includes *Fac-spec* **shows**
$\Gamma\vdash \{\!|\mathsf{m} = 3 \wedge \mathsf{i} = 2|\!\}$ r := **CALL** *Fac* (m) $\{\!|\mathsf{r} = 6 \wedge \mathsf{i} = 2|\!\}$

apply *vcg*

1. $\bigwedge i\ m.\ [\![m = 3; i = 2]\!] \Longrightarrow fac\ m = 6 \wedge i = 2$

In the verification condition the result variable r is replaced by *fac m*, which comes from the specification *Fac-spec*:

$$\forall n.\ \Gamma\vdash \{\!|\mathsf{n} = n|\!\}\ \mathsf{r} := \textbf{PROC}\ Fac(\mathsf{n})\ \{\!|\mathsf{r} = fac\ n|\!\}$$

In the resulting proof obligation, the universally quantified variable *n* is instantiated with the inital value of program variable m.

As the verification condition generator encounters a procedure call, for instance $\Gamma,\Theta\vdash P\ call\ init\ p\ return\ result\ Q,A$, it does not look inside the procedure body, but instead uses the specification $\forall Z.\ \Gamma,\Theta\vdash (P'\ Z)\ Call\ p\ (Q'\ Z),(A'\ Z)$ of the procedure. It adapts the specification to the actual calling context by a variant of a Kleymann-style consequence rule (cf. Section 3.1.1), which also takes parameter and result passing into account, similar to the rule for *block* introduced in the previous section. In the factorial example *n* plays the role of the auxiliary variable *Z*. It transports state information from the pre- to the postcondition.

$$\frac{\begin{array}{c}P \subseteq \{s.\ \exists Z.\ init\ s \in P'\ Z \wedge (\forall t{\in}Q'\ Z.\ return\ s\ t \in R\ s\ t) \wedge (\forall t{\in}A'\ Z.\ return\ s\ t \in A)\}\\ \forall s\ t.\ \Gamma,\Theta\vdash_{/F} (R\ s\ t)\ (result\ s\ t)\ Q,A \qquad \forall Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ (Call\ p)\ (Q'\ Z),(A'\ Z)\end{array}}{\Gamma,\Theta\vdash_{/F} P\ (call\ init\ p\ return\ result)\ Q,A}$$

The idea of this rule is to adapt the specification of *Call p* to *call init p return result*. Figure 4.2 shows the sequence of intermediate states for normal termination and how they are related to the assertions.

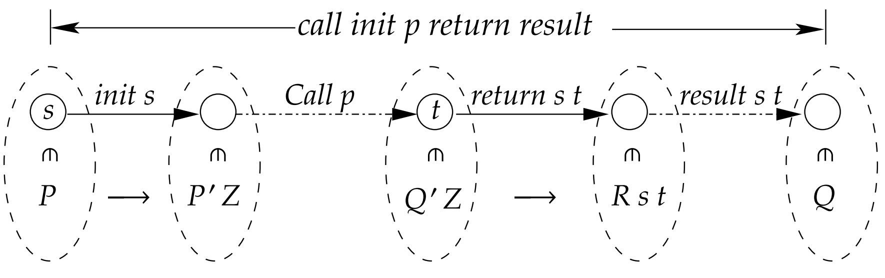

Figure 4.2: Procedure call and specification

We start in state *s* for which the precondition *P* holds. To be able to make use of the procedure specification we have to find a suitable instance of the auxiliary variable *Z*, such that the precondition of the specification holds: $init\ s \in P'\ Z$. Let *t* be the state immediately after execution of the procedure body, before returning to the caller and passing results. We know from the specification that the postcondition holds: $t \in Q'\ Z$. From this we have to conclude that leaving the procedure according to the function *return* leads to a state in *R s t*. From this state, execution of *result s t* ends up in a state in *Q*. For abrupt termination the analogous idea applies, but without the intermediate assertion *R s t*, since execution of *result s t* is skipped. Rewriting the record updates and selections in the side-condition with Isabelle's simplifier yields the natural proof obligation we have seen in the factorial example.

This automatic adaptation of the specification to the actual calling context is the reason why we use a variant of the Kleymann-style consequence rule instead of the more general consequence rule CONSEQ (cf. Figure 3.1 on p. 40). Technically such a rule would work for the verification condition generator as well, but it is less constructive. The user has to instantiate the specification by hand in the resulting verification condition.

The auxiliary variable Z has a polymorphic type. When a specification uses more than one auxiliary variable the verification condition generator instantiates Z with a tuple containing all the necessary auxiliary variables. Thereby a customised rule for the current specification is created on the fly.

Procedure Implementation — Partial Correctness To verify the procedure body we use the rule for recursive procedures. We extend the context with the procedure specification. In this extended context the specification holds by the assumption rule. We then verify the procedure body by using *vcg*, which uses the assumption to handle the recursive call.

lemma includes *Fac-impl* **shows**
$\forall n.\ \Gamma\vdash\{\!|\mathsf{n} = n|\!\}\ \mathsf{r} := \textbf{PROC}\ \mathit{Fac}(\mathsf{n})\ \{\!|\mathsf{r} = \mathit{fac}\ n|\!\}$
apply (*hoare-rule ProcRec1*)

```
1. ∀n. Γ,(⋃ₙ {({|n = n|}, "Fac", {|r = fac n|}, {})})
      ⊢ {|n = n|}
        IF n = 0 THEN r := 1
        ELSE r := CALL Fac(n − 1); r := n ∗ r FI
        {|r = fac n|}
```

apply *vcg*

1. $\bigwedge n.\ (n = 0 \longrightarrow 1 = \mathit{fac}\ n) \wedge (n \neq 0 \longrightarrow n * \mathit{fac}\ (n - 1) = \mathit{fac}\ n)$

The rule *ProcRec1* is a specialised version of the general rule for recursion (cf. Section 3.1.2), tailored for one recursive procedure. The method *hoare-rule* applies a single rule and solves canonical side-conditions like $p \in \mathit{dom}\ \Gamma$. Moreover, it expands the procedure body.

Procedure Implementation — Total Correctness For total correctness the user supplies a well-founded relation. For the factorial the input parameter n decreases in the recursive call. This is expressed by the measure function $\lambda(s,p).\ {}^{s}\mathsf{n}$. A state as prefix superscript, like ${}^{s}\mathsf{n}$, is syntactic sugar for $n\ s$. It is generally used in assertions and annotations to refer to the value of a variable at a certain state. The well-founded relation can depend on both the state space s and the procedure name p. The latter is useful to handle mutual recursion. The rule $\mathit{ProcRec1}_t$ is a specialised version of the general recursion rule for total correctness, tailored for one recursive procedure.

lemma includes *Fac-impl* **shows**
$\forall n.\ \Gamma\vdash_t \{\!|\mathsf{n} = n|\!\}\ \mathsf{r} := \textbf{PROC}\ \mathit{Fac}(\mathsf{n})\ \{\!|\mathsf{r} = \mathit{fac}\ n|\!\}$
apply (*hoare-rule* $\mathit{ProcRec1}_t$ [**where** *r=measure* ($\lambda(s,p).\ {}^{s}\mathsf{n}$)])

1. $\forall\sigma\ n.\ \Gamma,(\bigcup_n \{(\{\!|\mathsf{n} = n|\!\} \cap \{\!|\mathsf{n} < {}^{\sigma}\mathsf{n}|\!\}, \text{"}Fac\text{"}, \{\!|\mathsf{r} = fac\ n|\!\}, \{\})\})$
   $\vdash_t (\{\sigma\} \cap \{\!|\mathsf{n} = n|\!\})$
   IF $\mathsf{n} = 0$ **THEN** $\mathsf{r} := 1$ **ELSE** $\mathsf{r} :=$ **CALL** $Fac(\mathsf{n} - 1)$; $\mathsf{r} := \mathsf{n} * \mathsf{r}$ **FI**
   $\{\!|\mathsf{r} = fac\ n|\!\}$

We may only assume the specification for "smaller" states $\{\!|\mathsf{n} < {}^{\sigma}\mathsf{n}|\!\}$, where state σ is fixed in the precondition.

apply *vcg*

1. $\bigwedge n.\ (n = 0 \longrightarrow 1 = fac\ n) \wedge$
   $(n \neq 0 \longrightarrow n - 1 < n \wedge n * fac\ (n - 1) = fac\ n)$

The measure function results in the proof obligation $n - 1 < n$. In contrast to partial correctness we only assume "smaller" recursive procedure calls correct while verifying the procedure bodies. Were "smaller" is in the sense of a well-founded relation r.

The following contrived example was introduced by Homeier [50]. The only issue of this example is termination. It does not calculate anything interesting. We introduce two mutually recursive procedures.

```
procedures pedal(n,m) =
 IF 0 < n THEN
 IF 0 < m THEN
   CALL coast(n− 1,m− 1) FI; CALL pedal(n− 1,m)
 FI
and coast(n,m) =
 CALL pedal(n,m);
  IF 0 < m THEN CALL coast(n,m− 1) FI
```

The problem for termination is the call of *pedal* in procedure *coast*. If we only take the state space into account we cannot construct a proper well-founded relation that decreases with this call. Homeier introduces a call graph analysis in order to handle this example. In our setting the well-founded relation can also take the procedure names into account. We consider a call to *pedal* as progress with respect to procedure *coast*. We supply a measure function that weights *coast* as *1* and *pedal* as *0*. For each recursive call either n, m or the weight of the procedure name decreases. Since for the call of *coast* within *pedal* both n and m are decreased, the sum of n, m and the weight of the procedure name is sufficient as measure function.

lemma includes *pedal-coast-impl*
shows $\Gamma\vdash_t \{\!|True|\!\}$ **PROC** $pedal(\mathsf{n},\mathsf{m})$ $\{\!|True|\!\} \wedge \Gamma\vdash_t \{\!|True|\!\}$ **PROC** $coast(\mathsf{n},\mathsf{m})$ $\{\!|True|\!\}$
apply(*hoare-rule ProcRec2*$_t$
   [**where** r= *measure* $(\lambda(s,p).\ {}^{s}\mathsf{n} + {}^{s}\mathsf{m} + (\textit{if } p = \text{"}coast\text{"} \textit{ then } 1 \textit{ else } 0))$])

1. $\forall\sigma.\ \Gamma,\{(\{\!|\mathsf{n} + \mathsf{m} + 1 < {}^{\sigma}\mathsf{n} + {}^{\sigma}\mathsf{m}|\!\}, \text{"}coast\text{"}, \{\!|True|\!\}, \{\}),$
   $(\{\!|\mathsf{n} + \mathsf{m} < {}^{\sigma}\mathsf{n} + {}^{\sigma}\mathsf{m}|\!\}, \text{"}pedal\text{"}, \{\!|True|\!\}, \{\})\}$
   $\vdash_t \{\sigma\}$ **IF** $0 < \mathsf{n}$
      THEN IF $0 < \mathsf{m}$ **THEN CALL** $coast(\mathsf{n} - 1,\mathsf{m} - 1)$ **FI**; **CALL** $pedal(\mathsf{n} - 1,\mathsf{m})$
      FI
   $\{\!|True|\!\}$

2. $\forall \sigma.\ \Gamma,\{(\{\!|\mathsf{n} + \mathsf{m} < {}^{\sigma}\mathsf{n} + {}^{\sigma}\mathsf{m}|\!\}, \textit{"coast"}, \{\!|\textit{True}|\!\}, \{\}),$
$(\{\!|\mathsf{n} + \mathsf{m} < {}^{\sigma}\mathsf{n} + {}^{\sigma}\mathsf{m} + 1|\!\}, \textit{"pedal"}, \{\!|\textit{True}|\!\}, \{\})\}$
$\vdash_t \{\sigma\}$ **CALL** *pedal*(n,m); **IF** $0 <$ m **THEN CALL** *coast*(n,m − 1) **FI** $\{\!|\textit{True}|\!\}$

As the procedures are mutually recursive we also verify them simultaneously. This allows to augment the context with the specifications of both procedures. The rule *ProcRec2*$_t$ is derived from the general recursion rule CallRec to deal with two mutually recursive procedures. The specialisation of the CallRec Rule to a given number *n* of mutually recursive procedures is implemented as an Isabelle command.

In the example we get two subgoals, one for the body of *pedal* and one for the body of *coast*. In order to use the specification of *coast* in the body of *pedal*, the sum of n and m has to decrease at least by *2*. For the recursive call of *pedal* a progress of *1* is enough. In the body of *coast*, the sum of of n and m only has to decrease by *1* in case of a recursive call to *coast*. For *pedal* it can stay the same. Hence it is no problem to prove termination with this measure function.

4.5 Abrupt Termination

As explained in Section 2.4.4 we can implement breaking out of a loop by a **THROW** inside the loop body and enclosing the loop into a **TRY−CATCH** block. The following example again implements multiplication by iterated addition. This time the loop condition is always *True*. The actual test is implemented inside of the loop and exits the loop by a **THROW**.

```
lemma
Γ⊢ {|m = 0 ∧ s = 0|}
 TRY
  WHILE True
  INV {|s = m ∗ b|}
  DO IF m = a THEN THROW ELSE s := s + b; m := m + 1 FI OD
 CATCH
  SKIP
 END
{|s = a ∗ b|}
apply vcg
```

1. $\bigwedge m\ s.\ [\![m = 0; s = 0]\!] \Longrightarrow s = m * b$
2. $\bigwedge m\ s.\ [\![s = m * b; \textit{True}]\!] \Longrightarrow (m = a \longrightarrow s = a * b) \wedge (m \neq a \longrightarrow s + b = (m + 1) * b)$
3. $\bigwedge m\ s.\ [\![s = m * b; \neg\ \textit{True}]\!] \Longrightarrow s = a * b$

The first subgoal stems from the path from the precondition to the invariant. The second one from the loop body. We can assume the invariant and the loop condition and have to show that the invariant is preserved when we execute the **ELSE** branch, and that the **THEN** branch implies the assertion for abrupt termination, which is $\{\!|\mathsf{s} = a * b|\!\}$ according to the Rules Catch and Skip. The third subgoal expresses that normal termination of the while loop has to imply the postcondition. As loop never terminates normally the third subgoal trivially holds.

To model different kinds of abrupt termination (like `break`, `continue` or a procedure `return`), we introduce an additional variable Abr to the state space to distinguish them. Here is a modified version of the previous example that explicitly deals with this ghost variable.

```
lemma
Γ⊢ {|m = 0 ∧ s = 0|}
 TRY
  WHILE True
  INV {|s = m ∗ b|}
  DO IF m = a THEN Abr := ″Break″; THROW
    ELSE s := s + b; m := m + 1
    FI
  OD
 CATCH
   IF Abr = ″Break″ THEN SKIP ELSE THROW FI
 END
{|s = a ∗ b|}
apply vcg
```

```
 1. ⋀m s. [[m = 0; s = 0]] ⟹ s = m ∗ b
 2. ⋀m s. [[s = m ∗ b; True]]
              ⟹ (m = a ⟶
                    (″Break″ = ″Break″ ⟶ s = a ∗ b) ∧
                    (″Break″ ≠ ″Break″ ⟶ False)) ∧
                 (m ≠ a ⟶ s + b = (m + 1) ∗ b)
 3. ⋀m s. [[s = m ∗ b; ¬ True]] ⟹ s = a ∗ b
```

The proof obligation is basically the same as for the previous example. The second subgoal has the flaw to contain a trivial case distinction *″Break″ = ″Break″* and *″Break″ ≠ ″Break″*. It stems from the **IF** in the **CATCH** block that is propagated to the **THROW**. To avoid this blow up in the proof obligation the verification condition generator can be instrumented to simplify the proof obligation as it processes the assignment Abr := *″Break″*. At this point in the program it knows which value Abr gets. To distinguish this assignment from an ordinary assignment we introduce the command *raise*:

$$\begin{array}{l} \textit{raise} :: ('s \Rightarrow 's) \Rightarrow ('s,'p,'f)\ \textit{com} \\ \textit{raise}\ f \equiv \textit{Seq}\ (\textit{Basic}\ f)\ \textit{Throw} \end{array}$$

◀ Definition 4.2

As the verification condition generator encounters a *raise* command it invokes the simplifier. This avoids to produce odd case distinctions, as the following example illustrates:

```
lemma
Γ⊢ {|m = 0 ∧ s = 0|}
 TRY
  WHILE True
  INV {|s = m ∗ b|}
  DO IF m = a THEN RAISE Abr := ″Break″
    ELSE s := s + b; m := m + 1
    FI
  OD
 CATCH
   IF Abr = ''Break'' THEN SKIP ELSE THROW FI
 END
{|s = a ∗ b|}
apply vcg
```

1. $\bigwedge m\ s.\ [\![m = 0; s = 0]\!] \Longrightarrow s = m * b$
2. $\bigwedge m\ s.\ [\![s = m * b; \mathit{True}]\!] \Longrightarrow (m = a \longrightarrow s = a * b) \wedge (m \neq a \longrightarrow s + b = (m + 1) * b)$
3. $\bigwedge m\ s.\ [\![s = m * b; \neg\ \mathit{True}]\!] \Longrightarrow s = a * b$

In the context of a language with exceptions, this idea can also be employed to distinguish different kinds of exceptions like division by zero, dereferencing null pointers or lack of memory. Although the Hoare logic only provides a single postcondition for all kinds of abrupt termination, an invocation of the simplifier at the right point can be used to select the proper part of the postcondition.

4.6 Heap

The heap can contain structured values like `structs` in C or records in Pascal. We employ the split heap approach as described in Section 2.4.9.1. We have one heap variable f of type $\mathit{ref} \Rightarrow \mathit{value}$ for each component f of type *value* of the `struct`. References *ref* are isomorphic to the natural numbers and contain *NULL*.

A typical structure to represent a linked list in the heap is:

`struct list {int cont; struct list *next}`.

The structure contains two components, `cont` and `next`. So we also get two heap variables, *cont* of type $\mathit{ref} \Rightarrow \mathit{int}$ and *next* of type $\mathit{ref} \Rightarrow \mathit{ref}$ in our state space record:

	record *st* =
record *heap* =	*globals* :: *heap*
cont :: *ref* ⇒ *int*	*p* :: *ref*
next :: *ref* ⇒ *ref*	*q* :: *ref*
	r :: *ref*

To specify programs that manipulate the heap we follow the approach of Mehta and Nipkow [67]. We abstract the pointer structure in the heap to a suitable HOL type. A heap list is abstracted to a HOL lists of references. After this abstraction, specification and verification takes place in the domain of HOL lists. Predicate *List* abstracts the heap list to a HOL list:

Definition 4.3 ▸

$$\begin{aligned}
&\mathit{List} :: \mathit{ref} \Rightarrow (\mathit{ref} \Rightarrow \mathit{ref}) \Rightarrow \mathit{ref\ list} \Rightarrow \mathit{bool}\\
&\mathit{List}\ p\ h\ []\qquad = p = \mathit{NULL}\\
&\mathit{List}\ p\ h\ (a{\cdot}ps) = p = a \wedge p \neq \mathit{NULL} \wedge \mathit{List}\ (h\ p)\ h\ ps
\end{aligned}$$

The list of references is obtained from the heap h by starting with the reference p, following the references in h up to *NULL*. With a generalised predicate that describes a path in the heap, Mehta and Nipkow [68] show how this idea can canonically be extended to cyclic lists.

To specify in-place list reversal we can use the precondition $\{\!|$*List* p next *Ps*$|\!\}$ and the postcondition $\{\!|$*List* q next (*rev Ps*)$|\!\}$. Initially pointer *p* points to the list *Ps* and in the end pointer *q* points to the reverse of *Ps*. The following program implements this in-place list reversal and also shows the loop invariant.

```
Γ⊢ {|List p next Ps|}
    q := NULL;
    WHILE p ≠ NULL
    INV {|∃Ps' Qs'.
            List p next Ps' ∧
            List q next Qs' ∧ set Ps' ∩ set Qs' = {} ∧ rev Qs' @ Ps' = Ps|}
    DO r := p; p := p→next; r→next := q; q := r OD
    {|List q next (rev Ps)|}
```

The loop invariant expresses that q points to the already reversed initial part of the list and p to the not yet processed rest of the list. Moreover it asserts that lists *Ps'* and *Qs'* are separated.

For total correctness we also have to come up with a variant. In the loop of the example above, the pointer p is moved through the list. Hence the length of the list p points to is getting smaller in each iteration. In the invariant the list p points to is abstracted to *Ps'*, but *Ps'* is quantified inside the invariant. Hence we cannot directly refer to *Ps'* in the variant. However, the *List* predicate uniquely determines the list.

If List p h as and *List p h bs* then *as* = *bs*. ◂ Lemma 4.1

This uniqueness result can be exploited to convert the relational abstraction *List* to a functional abstraction:

$$list :: ref \Rightarrow (ref \Rightarrow ref) \Rightarrow ref\ list$$
$$list\ p\ h \equiv THE\ ps.\ List\ p\ h\ ps$$
◂ Definition 4.4

The definite description operator *THE* can be used instead of Hilbert's choice operator *SOME* in our case, since the value is unique. The following lemma connects *List* with *list*.

If List p h ps then *list p h* = *ps*. ◂ Lemma 4.2

With the functional abstraction *list* we can directly define the variant as the length of the list pointed to by p:

```
Γ⊢_t {|List p next Ps|}
    q := NULL;
    WHILE p ≠ NULL
    INV {|∃Ps' Qs'.
            List p next Ps' ∧
            List q next Qs' ∧ set Ps' ∩ set Qs' = {} ∧ rev Qs' @ Ps' = Ps|}
    VAR MEASURE |list p next|
    DO r := p; p := p→next; r→next := q; q := r OD
    {|List q next (rev Ps)|}
```

If we encapsulate in-place list reversal in a procedure it appears that the above specification is too weak. The problem is that we cannot determine from the Hoare triple what "does not change". This issue is referred to as *frame problem* in the literature [16]. In the context of the split-heap model we want to be able derive that all other heaps like cont have not changed. Moreover, for the next heap the only references that are affected are those in list *Ps*. The following definition illustrates how we approach the frame problem.

```
procedures Rev(p|q) =
 q := Null;
 WHILE p ≠ Null
 DO r := p; p := p→next; r→next := q; q := r OD

Rev-spec:
 ∀σ Ps. Γ⊢ {|σ. List p next Ps|} q := PROC Rev(p)
            {|List q next (rev Ps) ∧ (∀p. p ∉ set Ps ⟶ (next p = σnext p))|}
Rev-modifies:
 ∀σ. Γ⊢/UNIV {σ} q := PROC Rev(p) {t. t may-only-modify-globals σ in [next]}
```

We give two specifications this time. The first one captures the functional behaviour and additionally expresses that all parts of the *next*-heap not contained in *Ps* remain the same (σ denotes the pre-state). Fixing a state is part of the assertion syntax: {|σ. ...|} translates to {*s. s*=σ ...} and $^{\sigma}$next to *next* (*globals* σ). The second specification is a "modifies-clause" or "frame condition" that lists all the state components that may be changed by the procedure. Only the *next* heap is listed, and therefore we know that the *cont* heap or any other heap remains unchanged. Thus the main specification can focus on the relevant parts of the state space. The assertion *t may-only-modify-globals σ in* [*next*] abbreviates the following relation between the final state *t* and the initial state σ: ∃*next'*. *globals t* = (*globals* σ)(|*next*:=*next'*|). This frame condition can be exploited during verification condition generation:

```
lemma includes Rev-spec + Rev-modifies shows
Γ⊢{|cont=c ∧ List p next Ps|} p := CALL Rev(p)
   {|cont=c ∧ List p next (rev Ps)|}
 apply vcg

 1. ⋀next cont p.
       List p next Ps ⟹
       ∀nexta q.
         List q nexta (rev Ps) ∧ (∀p. p ∉ set Ps ⟶ nexta p = next p) ⟶
         cont = cont ∧ List q nexta (rev Ps)
```

The impact of the frame condition shows up in the resulting proof obligation. The *cont*-heap results in the same variable before and after the procedure call (*cont* = *cont*), whereas the *next*-heap is described by *next* in the beginning and by *nexta* in the end. The specification of *Rev* relates both *next*-heap states.

So how does the verification condition generator use the modifies-clause to obtain this result? A procedure call is of the form *call init p return result*. The return function is defined as *return s t* = *s*(|*globals* := *globals t*|). It copies the current state of the global variables (including the heap) at the end of the procedure body to the caller. The verification condition generator exploits the modifies-clause to substitute this return function with a modified one, which only returns the global components that may actually change. In the example this is:

$$\textit{return}'\ s\ t = s(\!|\textit{globals} := (\textit{globals}\ s)(\!|\textit{next} := \textit{next}\ (\textit{globals}\ t)|\!)|\!).$$

So *cont* actually behaves like a local variable in the resulting proof obligation. The Hoare rule that justifies this modification of the *return* function is the following.

$$\frac{\begin{array}{c}\Gamma,\Theta\vdash_{/F} P\ \textit{call init p return}'\ \textit{result}\ Q,A\\ \forall s\ t.\ t \in \textit{Modif}\ (\textit{init}\ s) \longrightarrow \textit{return}'\ s\ t = \textit{return}\ s\ t\\ \forall s\ t.\ t \in \textit{ModifAbr}\ (\textit{init}\ s) \longrightarrow \textit{return}'\ s\ t = \textit{return}\ s\ t\\ \forall \sigma.\ \Gamma,\Theta\vdash_{/\textit{UNIV}} \{\sigma\}\ \textit{Call}\ p\ (\textit{Modif}\ \sigma),(\textit{ModifAbr}\ \sigma)\end{array}}{\Gamma,\Theta\vdash_{/F} P\ \textit{call init p return result}\ Q,A}\ (\textsc{ModifyReturn})$$

The last premise is the modifies-clause. It has a postcondition *Modif* σ for normal termination and *ModifAbr* σ for abrupt termination. In case of the list reversal, the modifies-clause for abrupt termination is the empty set, since the procedure never terminates abruptly. State σ is the initial state. If the functions *return'* and *return* behave the same under the assumption of the modifies-clause then it is valid to replace them. In our example we have to show the following side-condition. Note that *init s* does not modify the global components at all and hence we have the equation *globals* (*init s*) = *globals s*.

$$\begin{array}{l}\forall s\ t.\ t \in \{t.\ \exists \textit{next}'.\ \textit{globals}\ t = (\textit{globals}\ s)(\!|\textit{next} := \textit{next}'|\!)\} \longrightarrow\\ \quad s(\!|\textit{globals} := (\textit{globals}\ s)(\!|\textit{next} := \textit{next}\ (\textit{globals}\ t)|\!)|\!) = s(\!|\textit{globals} := \textit{globals}\ t|\!)\end{array}$$

We have to show the equality of two records. Two records are equal if all their components are equal. For component *next* both sides yield the *next'* that is obtained from the premise. All other components are from state *s*. These side-conditions are solved automatically by the verification condition generator.

For the modifies-clause in the ModifyReturn Rule, the fault set is *UNIV* instead of *F* in the rest of the rule. This means that we can ignore guards while verifying the modifies-clause itself. This is sound, since the first premise already takes care of the guards. The modifies-clause is proven fully automatically by the verification condition generator. In case of a loop, the invariant is the modifies-clause itself. To handle procedure calls the verification condition generator uses the corresponding modifies-clause as specification. The resulting verification condition is similar to the side-condition of the ModifyReturn Rule described above and can be solved in the same fashion.

Also in the context of total correctness the modify-clause only has to be proven for partial correctness. Termination of the procedure call is already handled by the first premise.

$$\frac{\begin{array}{c}\Gamma,\Theta\vdash_{t/F} P\ \textit{call init p return}'\ \textit{result}\ Q,A\\ \forall s\ t.\ t \in \textit{Modif}\ (\textit{init}\ s) \longrightarrow \textit{return}'\ s\ t = \textit{return}\ s\ t\\ \forall s\ t.\ t \in \textit{ModifAbr}\ (\textit{init}\ s) \longrightarrow \textit{return}'\ s\ t = \textit{return}\ s\ t\\ \forall \sigma.\ \Gamma,\Theta\vdash_{/\textit{UNIV}} \{\sigma\}\ \textit{Call}\ p\ (\textit{Modif}\ \sigma),(\textit{ModifAbr}\ \sigma)\end{array}}{\Gamma,\Theta\vdash_{t/F} P\ \textit{call init p return result}\ Q,A}$$

The modifies-clause lifts the advantages of the split-heap approach to the level of procedure specifications. It is an effective way to express separation of different pointer structures in the heap and can be handled completely automatically during verification condition generation. For example, reversing a list only modifies the *next*-heap but not some *left*- and *right*-heaps of a tree structure.

4.7 Dynamic Procedure Call

The Hoare rule for the dynamic command *DynCom* is the following.

$$\frac{\forall s{\in}P.\ \Gamma,\Theta\vdash_{/F} P\ (c_s\ s)\ Q,A}{\Gamma,\Theta\vdash_{/F} P\ (DynCom\ c_s)\ Q,A}$$

As command c_s depends on the state we have to show the Hoare triple for all possible states that satisfy the precondition. We can use the consequence rule to transform this rule to a form that the verification condition generator can use:

$$\frac{P \subseteq \{s.\ \exists P'\ Q'\ A'.\ \Gamma,\Theta\vdash_{/F} P'\ (c_s\ s)\ Q',A' \wedge P \subseteq P' \wedge Q' \subseteq Q \wedge A' \subseteq A\}}{\Gamma,\Theta\vdash_{/F} P\ (DynCom\ c_s)\ Q,A}$$

For every possible state s in P we have to come up with a suitable specification $\Gamma,\Theta\vdash_{/F} P'\ (c_s\ s)\ Q',A'$. If we do not put any restrictions on command c_s this is the best we can expect for this general state dependent command. However, for specific applications of the dynamic command it might be possible for the verification condition generator to infer the specification from the context, for example, consider procedure pointers. A typical application of a procedure pointer is to pass the comparison function as an argument to a sorting procedure. The sorting procedure does not modify this pointer but just uses it to call the procedure. Hence the specification of the comparison function can be fixed throughout the sorting procedure. To simplify the example we use a procedure that calculates the maximum of two numbers instead of the sorting procedure.

First, we define the signature of the comparison function:

procedures *compare*(*n*,*m*|*b*)

This declaration is only used to generate the syntax translations for (dynamic) procedure calls to *compare*.

```
procedures Max (compare, n, m | k) =
b := DYNCALL compare(n,m);
IF b THEN k := n ELSE k := m FI
```

Max-spec:

```
 ∀leq n m. Γ⊢
   (⦃n=n ∧ m=m⦄ ∩
    ⦃∀n m. Γ⊢ ⦃n=n ∧ m=m⦄ b := PROC compare(n,m) ⦃b = (leq n m)⦄⦄)
    k := PROC Max(compare,n,m)
    ⦃k = mx leq n m⦄
```

First, compare is an ordinary program variable as n or m. Its type is *string*, since procedure names are represented as strings. The dynamic procedure call b := **DYNCALL** compare(n,m) is translated to

$$dynCall\ (\lambda s.\ s(\!|n := n\ s,\ m := m\ s|\!))\ compare\ (\lambda s\ t.\ s(\!|globals := globals\ t|\!))\ (\lambda i\ t.\ \mathsf{b} := b\ t)$$

where *compare* is the selector function of the state space record. The precondition of the specification of *Max* consists of two parts that are conjoined by the intersection. The first part fixes the initial values of n and m and the second part contains the expected behaviour of the procedure compare, specified as a Hoare triple. As the assertions can contain arbitrary HOL expressions it is no problem to include a Hoare triple in a pre- or postcondition. Note that b := **PROC** compare(n,m) in the precondition translates to *Call* (*compare s*), where *s* is the implicitly bound state of the precondition. Hence the Hoare triple in the precondition describes the behaviour of the procedure that is referenced by the initial value of compare. As we do not know the exact behaviour of the comparison function the postconditions are parametrised by the place-holder *leq* of type *nat* ⇒ *nat* ⇒ *bool*. Function *mx* is defined as follows:

$$mx\ leq\ a\ b \equiv \mathsf{if}\ leq\ a\ b\ \mathsf{then}\ a\ \mathsf{else}\ b$$

The verification condition generator takes the specification in the precondition and applies it to the last premise of the following rule, as it processes the dynamic procedure call:

$$\frac{\begin{array}{c}P \subseteq \{s.\ p\ s = q \wedge (\exists Z.\ init\ s \in P'\ Z \wedge (\forall t{\in}Q'\ Z.\ return\ s\ t \in R\ s\ t) \wedge (\forall t{\in}A'\ Z.\ return\ s\ t \in A))\}\\ \forall s\ t.\ \Gamma,\Theta\vdash_{/F} (R\ s\ t)\ (c\ s\ t)\ Q,A \qquad \forall Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ (Call\ q)\ (Q'\ Z),(A'\ Z)\end{array}}{\Gamma,\Theta\vdash_{/F} P\ (dynCall\ init\ p\ return\ c)\ Q,A}$$

This rule resembles the rule for ordinary procedure calls as introduced in Section 4.4. Additionally we have to show that $p\ s = q$, where $p\ s$ is the procedure name that is actually called and q the fixed procedure name of the specification. In the example $p\ s = q$ ensures that the variable compare still holds the same value as in the initial state.

To extract the embedded Hoare Triple from the precondition of *Max* we use the following variant of the consequence rule.

$$\frac{\forall s{\in}S.\ \Gamma,\Theta\vdash (\{s\} \cap P)\ c\ Q,A}{\Gamma,\Theta\vdash (P \cap S)\ c\ Q,A}\ (\textsc{ConseqExtractPre})$$

This rule is applied backwards. It allows to bring the precondition *S* in front of the Hoare triple, while fixing the state.

lemma *Max-spec*: **includes** *Max-impl*
shows
∀*n m leq*. Γ⊢
  (⦃n=*n* ∧ m=*m*⦄ ∩
  ⦃∀*n' m'*. Γ⊢ ⦃n=*n'* ∧ m=*m'*⦄ b := **PROC** compare(n,m) ⦃b = (*leq n' m'*)⦄⦄)
  k := **PROC** *Max*(compare,n,m)
  ⦃k = *mx leq n m*⦄
apply (*hoare-rule ProcNoRec1*)

1. ⋀*n m leq*.
      Γ⊢ (⦃n = *n* ∧ m = *m*⦄ ∩
            ⦃∀*n' m'*. Γ⊢ ⦃n = *n'* ∧ m = *m'*⦄ b := **PROC** compare(n,m) ⦃b = *leq n' m'*⦄⦄)
          b := **DYNCALL** compare(n,m); **IF** b **THEN** k := n **ELSE** k := m **FI**
          ⦃k = *mx leq n m*⦄

apply (*rule ConseqExtractPre*)

1. $\bigwedge n\ m\ leq.$
   $\forall s \in \{\!|\forall n'\ m'.\ \Gamma \vdash \{\!|\mathsf{n} = n' \wedge \mathsf{m} = m'|\!\}\ \mathsf{b} := \textbf{PROC}\ \mathsf{compare(n,m)}\ \{\!|\mathsf{b} = leq\ n'\ m'|\!\}|\!\}.$
   $\Gamma \vdash (\{s\} \cap \{\!|\mathsf{n} = n \wedge \mathsf{m} = m|\!\})$
   $\mathsf{b} := \textbf{DYNCALL}\ \mathsf{compare(n,m)};\ \textbf{IF}\ \mathsf{b}\ \textbf{THEN}\ \mathsf{k} := \mathsf{n}\ \textbf{ELSE}\ \mathsf{k} := \mathsf{m}\ \textbf{FI}$
   $\{\!|\mathsf{k} = mx\ leq\ n\ m|\!\}$

apply *clarify*

1. $\bigwedge n\ m\ leq\ s.$
   $\forall n'\ m'.\ \Gamma \vdash \{\!|\mathsf{n} = n' \wedge \mathsf{m} = m'|\!\}\ Call\ (compare\ s)\ \{\!|\mathsf{b} = leq\ n'\ m'|\!\} \Longrightarrow$
   $\Gamma \vdash (\{s\} \cap \{\!|\mathsf{n} = n \wedge \mathsf{m} = m|\!\})$
   $\mathsf{b} := \textbf{DYNCALL}\ \mathsf{compare(n,m)};\ \textbf{IF}\ \mathsf{b}\ \textbf{THEN}\ \mathsf{k} := \mathsf{n}\ \textbf{ELSE}\ \mathsf{k} := \mathsf{m}\ \textbf{FI}$
   $\{\!|\mathsf{k} = mx\ leq\ n\ m|\!\}$

apply *vcg*

1. $\bigwedge leq\ compare\ n\ m.$
   $\forall n'\ m'.\ \Gamma \vdash \{\!|\mathsf{n} = n' \wedge \mathsf{m} = m'|\!\}\ Call\ compare\ \{\!|\mathsf{b} = leq\ n'\ m'|\!\} \Longrightarrow$
   $(leq\ n\ m \longrightarrow n = mx\ leq\ n\ m) \wedge (\neg\ leq\ n\ m \longrightarrow m = mx\ leq\ n\ m)$

apply (*clarsimp simp add*: *mx-def*)
done

Rule *ProcNoRec1* is a variant of the recursion rule for the case were no recursion occurs. It only expands the body. The next two steps are used to extract the specification out of the precondition. Note that the state *s* is introduced and the value of compare becomes *compare s*. Then the verification condition generator can be called and uses the specification as it processes the dynamic procedure call.

We can now implement a concrete comparison:

procedures *LEQ* (*n,m* | *b*) = b := n ≤ m

We prove the specification:

$$\forall n\ m.\ \Gamma \vdash \{\!|\mathsf{n} = n \wedge \mathsf{m} = m|\!\}\ \mathsf{b} := \textbf{PROC}\ LEQ\mathsf{(n,m)}\ \{\!|\mathsf{b} = (n \le m)|\!\}$$

Then we can derive the specialised specification for the instantiated maximum procedure:

$$\forall n\ m.\ \Gamma \vdash \{\!|\mathsf{n} = n \wedge \mathsf{m} = m|\!\}\ \mathsf{k} := \textbf{CALL}\ Max(''LEQ'',\mathsf{n,m})\ \{\!|\mathsf{k} = mx\ (\le)\ n\ m|\!\}$$

These ideas to handle a dynamic procedure call can also be adapted to an object oriented setting for dynamic method invocation. The method called, depends on the dynamic type of the object. A type annotation in the program can give the verification condition generator a hint, which specification to select. This could be a specification that subsumes the behaviour of the methods for all subtypes as well. The user then has to show that the actual type fits to the one in the hint. The selection of the specification may even be easier as in case of the procedure pointer example, as it does not have to be part of the precondition of a method. The specifications are grouped according to the class hierarchy and are thus static. Only the type of the objects is dynamic. Some type constraint for an object is sufficient to determine which specifications may be relevant.

As the example of the procedure pointer illustrates it is possible to customise the verification condition generator to commonly used patterns of the dynamic procedure call. As default solution one can always use the generic rule introduced in the beginning of this section. This rule postpones the selection of the relevant specification for the dynamic procedure call to the user. This may be the only sufficient solution in a sophisticated program that passes around and calculates procedure pointers.

4.8 Closures

Dealing with closures is quite similar to dynamic procedure calls. Additionally closures can be used to hide parts of the state. For example, here is the implementation of a private counter in ML:

```
fun inc p () = p := !p + 1; !p

fun newCounter () =
    let
      val p = ref 0
    in inc p
    end
```

Dereferencing p is written as !p in ML. First a new counter is created by allocating an integer and initialising it with 0. The address p of this new location is passed to function inc as partial application. Since p is a local name inside newCounter nobody else knows this address. The result of newCounter is a function of type unit => nat. Every time we apply it to () it increments the private counter and returns its value. For example,

```
val count = newCounter (); val x = count () + count ().
```

The value of x is 3.

We now implement the same program in Simpl. A counter consist of a single field cnt in the heap.

procedures *Inc*(*p*|*r*) =
p→cnt := p→cnt + *1*;
r := p→cnt

The specification of *Inc* is the following.

lemma (**in** *Inc-impl*)
$\forall i\, p.\ \Gamma \vdash \{\!|$p→cnt = $i|\!\}$ r := **PROC** *Inc*(p) $\{\!|$r=i+1 ∧ p→cnt = i+$1|\!\}$
by *vcg simp*

Next, we define the procedure *NewCounter*. It allocates a new counter cell, initialises its value to *0* and creates a closure for procedure *Inc*. The *NEW* is implemented as described in Section 2.4.9.2. The first argument is the size of the new cell. It checks whether there is enough free memory left and creates a *new* reference, or otherwise returns *NULL*. The partial application is implemented as described in Section 2.4.7.

procedures *NewCounter*(|*c*) =
p := *NEW 1* [cnt := *0*];
c := ([(*"p"*,p)],*"Inc"*)

How does a proper specification for *NewCounter* look like? Of course we could just reveal everything and expose the content of the closure in the postcondition. However, a closure should be viewed as a black box, at least when we want to deal with higher order procedures in a modular fashion. As we specify procedures by the parameterless procedure call **PROC**, we use the parameterless closure call *callClosure* to specify a closure. The auxiliary function *upd* is defined analogously to the example in Section 2.4.7:

$$\exists p.\ \forall i.\ \Gamma\vdash \{\!|p{\rightarrow}\mathsf{cnt} = i|\!\}\ \mathit{callClosure\ upd\ c}\ \{\!|\mathsf{r} = i + 1 \wedge p{\rightarrow}\mathsf{cnt} = i + 1|\!\}.$$

There is a reference *p* such that a call to the closure increments the corresponding counter. The hidden reference is existentially quantified. Of course we should also provide a frame condition, but as this works analogously to the examples in Section 4.6 we omit it here. Next comes the specification for *NewCounter*. The ghost variable alloc is a list of allocated references and free indicates how much memory is still left. Additionally to the Hoare triple about the resulting closure, we express that the reference is fresh and that the initial value of the counter is *0*. The freshness of the reference is crucial if we want to reason about several counters.

lemma (**in** *NewCounter-impl*)
shows *NewCounter-spec*:
$\forall$ *alloc free*.
$\Gamma\vdash \{\!|1 \le \mathsf{free} \wedge \mathsf{free}{=}\mathit{free} \wedge \mathsf{alloc}{=}\mathit{alloc}|\!\}$ c := **PROC** *NewCounter*()
  $\{\!|\exists p.\ p \notin \mathit{set\ alloc} \wedge p \in \mathit{set}\ \mathsf{alloc} \wedge \mathsf{free}{=}\mathit{free} - 1 \wedge p \neq \mathit{NULL} \wedge p{\rightarrow}\mathsf{cnt} = 0 \wedge$
    $(\forall i.\ \Gamma\vdash \{\!|p{\rightarrow}\mathsf{cnt} = i|\!\}\ \mathit{callClosure\ upd}\ \mathsf{c}\ \{\!|\mathsf{r}{=}i{+}1 \wedge p{\rightarrow}\mathsf{cnt} = i{+}1|\!\})|\!\}$
apply *vcg*
apply (*rule-tac x=new* (*set alloc*) **in** *exI*)
apply *simp*

1. $\bigwedge$*alloc free*.
   $1 \le \mathit{free} \Longrightarrow$
   $\forall i.\ \Gamma\vdash \{\!|\mathit{new}\ (\mathit{set\ alloc}){\rightarrow}\mathsf{cnt} = i|\!\}$
     callClosure upd ([(*"p"*, *new* (*set alloc*))], *"Inc"*)
     $\{\!|\mathsf{r} = i + 1 \wedge \mathit{new}\ (\mathit{set\ alloc}){\rightarrow}\mathsf{cnt} = i + 1|\!\}$

After applying the verification condition generator and instantiating reference *p* we end up with the Hoare triple from the postcondition. The c is substituted by the current closure. Since *callClosure* is only the composition of a *Basic* command and a *Call* this can be verified as usual by calling the vcg.

We continue with a (dynamic) call of the closure. As for the dynamic command, the default rule is an adaptation of the general consequence rule.

$$\frac{\begin{array}{l} P \subseteq \{s.\ \exists P'\ Q'\ A'.\\ \qquad \Gamma,\Theta\vdash_{/F} P'\ (\mathit{callClosure\ upd}\ (\mathit{cl}\ s))\ Q',A' \wedge\\ \qquad \mathit{init}\ s \in P' \wedge (\forall t{\in}Q'.\ \mathit{return}\ s\ t \in R\ s\ t) \wedge (\forall t{\in}A'.\ \mathit{return}\ s\ t \in A)\}\\ \qquad\qquad \forall s\ t.\ \Gamma,\Theta\vdash_{/F} (R\ s\ t)\ (c\ s\ t)\ Q,A \end{array}}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{dynCallClosure\ init\ upd\ cl\ return}\ c)\ Q,A}$$

Using this rule, the verification condition generator postpones the selection of proper specification of the closure to the user.

lemma
$\Gamma\vdash\{\!|\exists p.\ p{\rightarrow}\mathsf{cnt} = i \wedge (\forall i.\ \Gamma\vdash \{\!|p{\rightarrow}\mathsf{cnt} = i|\!\}\ \mathit{callClosure\ upd}\ \mathsf{c}\ \{\!|\mathsf{r}{=}i{+}1 \wedge p{\rightarrow}\mathsf{cnt} = i{+}1|\!\})|\!\}$
$\quad \mathit{dynCallClosure}\ (\lambda s.\ s)\ \mathit{upd}\ c\ (\lambda s\ t.\ s(\!|\mathit{globals} := \mathit{globals}\ t|\!))\ (\lambda s\ t.\ \mathit{Basic}\ (\lambda u.\ u(\!|r := r\ t|\!)))$
$\quad \{\!|\mathsf{r}{=}i{+}1|\!\}$

apply vcg

$$\begin{array}{l} 1.\ \bigwedge s\ p.\ \forall i.\ \Gamma\vdash \{\!|p{\rightarrow}\mathsf{cnt} = i|\!\}\ \mathit{callClosure\ upd}\ (c\ s)\\ \qquad\qquad \{\!|\mathsf{r} = i + 1 \wedge p{\rightarrow}\mathsf{cnt} = i + 1|\!\} \Longrightarrow\\ \qquad \exists P'\ Q'.\\ \qquad\quad \Gamma\vdash P'\ \mathit{callClosure\ upd}\ (c\ s)\ Q' \wedge\\ \qquad\quad s \in P' \wedge (\forall t{\in}Q'.\ r\ t = \mathit{cnt}\ (\mathit{globals}\ s)\ p + 1) \end{array}$$

Now we can employ the specification in the assumption for $i = \mathit{cnt}\ (\mathit{globals}\ s)\ p$ and finally instantiate P' and Q' to finish the proof.

We can apply the same idea as for the dynamic procedure call in Section 4.7 so that the verification condition generator already selects the specification. We first extract the specification from the precondition so that the verification condition generator can guess it. We have to show that the current closure is the same as the one in the specification:

$$\frac{\begin{array}{l} P \subseteq \{s.\ \exists Z.\ \mathit{cl}' = \mathit{cl}\ s \wedge\\ \qquad \mathit{init}\ s \in P'\ Z \wedge (\forall t{\in}Q'\ Z.\ \mathit{return}\ s\ t \in R\ s\ t) \wedge (\forall t{\in}A'\ Z.\ \mathit{return}\ s\ t \in A)\}\\ \qquad\qquad \forall s\ t.\ \Gamma,\Theta\vdash_{/F} (R\ s\ t)\ (c\ s\ t)\ Q,A\\ \qquad \forall Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ (\mathit{callClosure\ upd\ cl}')\ (Q'\ Z),(A'\ Z) \end{array}}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{dynCallClosure\ init\ upd\ cl\ return}\ c)\ Q,A}$$

The following example illustrates this approach.

lemma
$\Gamma\vdash\{\!|\exists p.\ p{\rightarrow}\mathsf{cnt} = i \wedge (\forall i.\ \Gamma\vdash \{\!|p{\rightarrow}\mathsf{cnt} = i|\!\}\ \mathit{callClosure\ upd}\ \mathsf{c}\ \{\!|\mathsf{r}{=}i{+}1 \wedge p{\rightarrow}\mathsf{cnt} = i{+}1|\!\})|\!\}$
$\quad \mathit{dynCallClosure}\ (\lambda s.\ s)\ \mathit{upd}\ c\ (\lambda s\ t.\ s(\!|\mathit{globals} := \mathit{globals}\ t|\!))\ (\lambda s\ t.\ \mathit{Basic}\ (\lambda u.\ u(\!|r := r\ t|\!)))$
$\quad \{\!|\mathsf{r}{=}i{+}1|\!\}$
apply (*rule ConseqExtractPre′*)
apply *clarify*

$$\begin{array}{l} 1.\ \bigwedge s\ p.\ \forall i.\ \Gamma\vdash \{\!|p{\rightarrow}\mathsf{cnt} = i|\!\}\ \mathit{callClosure\ upd}\ (c\ s)\\ \qquad\qquad \{\!|\mathsf{r} = i + 1 \wedge p{\rightarrow}\mathsf{cnt} = i + 1|\!\} \Longrightarrow\\ \qquad \Gamma\vdash \{s\}\\ \qquad\quad \mathit{dynCallClosure}\ (\lambda s.\ s)\ \mathit{upd}\ c\ (\lambda s\ t.\ s(\!|\mathit{globals} := \mathit{globals}\ t|\!))\\ \qquad\quad (\lambda s\ t.\ \mathsf{r} := r\ t) \end{array}$$

$\{\!|\mathsf{r} = {}^s\mathsf{cnt}\ p + 1|\!\}$

apply vcg

$$\begin{array}{l}1.\ \bigwedge p\ cnt\ c.\\ \quad \forall i.\ \Gamma\vdash \{\!|p\rightarrow\mathsf{cnt} = i|\!\}\ callClosure\ upd\ c\\ \qquad \{\!|\mathsf{r} = i + 1 \wedge p\rightarrow\mathsf{cnt} = i + 1|\!\} \Longrightarrow\\ \quad \forall cnta\ r.\ r = cnt\ p + 1 \wedge cnta\ p = cnt\ p + 1 \longrightarrow r = cnt\ p + 1\end{array}$$

by *simp*

The final example for counters introduces aliasing between the counters in closures c and d. Both refer to the same counter and hence the result is *3*. Since closures are ordinary values, the assignment d := c makes them equal and hence we can use the specification from the postcondition of *NewCounter* to handle both calls.

lemma (**in** *NewCounter-impl*) **shows**
$\Gamma\vdash\{\!|1 \le \mathsf{free}|\!\}$
  c := **CALL** *NewCounter* ();
  d := c;
  dynCallClosure ($\lambda s.\ s$) *upd c* ($\lambda s\ t.\ s(\!|globals := globals\ t|\!)$) ($\lambda s\ t.\ Basic\ (\lambda u.\ u(\!|n := r\ t|\!))$);
  dynCallClosure ($\lambda s.\ s$) *upd d* ($\lambda s\ t.\ s(\!|globals := globals\ t|\!)$) ($\lambda s\ t.\ Basic\ (\lambda u.\ u(\!|m := r\ t|\!))$);
  r := n + m
  $\{\!|\mathsf{r}=3|\!\}$

The next section explains how annotations can be inserted into a program text, to guide the verification condition generation. This technique can also be used to annotate the calling points of closures so that the verification condition generator can guess the proper specification, instead of postponing the decision to the user. The verification condition generator is implemented as a ML tactic. Hence it can inspect the Hoare triple to look for a proper specification. It can even employ the approach to extract the triple from the precondition to discharge the side-condition about the specification.

Another aspect of closures is to partially apply a closure to some more arguments. In our representation of the environment as association list this means to augment the list.

Definition 4.5 ►

$$\begin{array}{l}ap :: ('k \times 'v)\ list \Rightarrow (('k \times 'v)\ list \times 'p) \Rightarrow (('k \times 'v)\ list \times 'p)\\ ap\ es\ cl \equiv (es\ @\ fst\ cl,\ snd\ cl)\end{array}$$

Consider a closure c which implements the addition of two parameters n and m. We view this closure as a black box. We do not make any assumptions about the environment in it. It could be empty or not. When we partially apply an argument to the closure, we expect the resulting closure to implement an increment function that expects only one argument, namely m. This is what the next example attempts to prove.

lemma
$\Gamma\vdash\{\!|\mathsf{n}=n_0 \wedge (\forall i\ j.\ \Gamma\vdash \{\!|\mathsf{n}=i \wedge \mathsf{m}=j|\!\}\ callClosure\ upd\ \mathsf{c}\ \{\!|\mathsf{r}=i + j|\!\})|\!\}$
  c := (*ap* [(''*n*'',n)] c)
  $\{\!|\forall j.\ \Gamma\vdash \{\!|\mathsf{m}=j|\!\}\ callClosure\ upd\ \mathsf{c}\ \{\!|\mathsf{r}=n_0 + j|\!\}|\!\}$

apply vcg

$$\begin{array}{l}1.\ \bigwedge s\ j.\ \forall i\ j.\ \Gamma\vdash \{\!|\mathsf{n} = i \wedge \mathsf{m} = j|\!\}\ callClosure\ upd\ (c\ s)\ \{\!|\mathsf{r} = i + j|\!\} \Longrightarrow\\ \qquad \Gamma\vdash \{\!|\mathsf{m} = j|\!\}\ callClosure\ upd\ (ap\ [(''n'',\ n\ s)]\ (c\ s))\ \{\!|\mathsf{r} = {}^s\mathsf{n} + j|\!\}\end{array}$$

The resulting proof obligation is quite similar to the adaptation of a procedure specification to an actual procedure call. It is even simpler since we only have to deal with parameter passing and not with a procedure return and result passing. The adaptation rule is hence again a variant of the consequence rule.

$$\frac{P \subseteq \{s.\ \exists P'\ Q'\ A'.\ \Gamma,\Theta\vdash_{/F} P'\ (\mathit{callClosure\ upd\ cl})\ Q',A' \wedge \mathit{upd\ es\ s} \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A\}}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{callClosure\ upd}\ (\mathit{ap\ es\ cl}))\ Q,A}$$

Or in the Kleymann-style:

$$\frac{\begin{array}{c} P \subseteq \{s.\ \exists Z.\ \mathit{upd\ es\ s} \in P'\ Z \wedge Q'\ Z \subseteq Q \wedge A'\ Z \subseteq A\} \\ \forall Z.\ \Gamma,\Theta\vdash_{/F} (P'\ Z)\ (\mathit{callClosure\ upd\ cl})\ (Q'\ Z),(A'\ Z)\end{array}}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{callClosure\ upd}\ (\mathit{ap\ es\ cl}))\ Q,A}$$

These rules are instances for our concrete model of environments as association list. The general rules, which only consider the environment to be of type *'e*, have a semantical side-condition on the different environments in the premise and the conclusion. In this abstract view we do not know how the environment is implemented and the *ap* function is not yet available. The rule just assumes that we switch from an environment e' to e. The side-condition $\mathit{upd}\ e = \mathit{upd}\ e' \circ \mathit{upd}\ x$ specifies semantically that the environment e is an extension of e'. The x models the new arguments that are partially applied. Note that here function *upd* is of type $'e \Rightarrow\ 's \Rightarrow\ 's$ and not the concrete update of the example above.

$$\frac{\begin{array}{c} P \subseteq \{s.\ \exists P'\ Q'\ A'.\ \Gamma,\Theta\vdash_{/F} P'\ (\mathit{callClosure\ upd}\ (e',\ p))\ Q',A' \wedge \mathit{upd}\ x\ s \in P' \wedge Q' \subseteq Q \wedge A' \subseteq A\} \\ \mathit{upd}\ e = \mathit{upd}\ e' \circ \mathit{upd}\ x\end{array}}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{callClosure\ upd}\ (e,\ p))\ Q,A}$$

The side-condition can be discharged in the implementation of the environment as an association list. Thats why it does not show up in the specialised rules before.

4.9 Introducing Annotations

When verifying a larger piece of program text, it is useful to split it and prove the parts in isolation. The parts can then be recombined with the consequence rule. Moreover, it should be possible to refer to an intermediate state in annotations like a loop invariant. To automate this process we introduce the derived command *specAnno*, which allows to introduce a Hoare triple (including auxiliary variables) in the program text.

◀ Definition 4.6

$\mathit{specAnno}::('a\Rightarrow's\ set)\Rightarrow('a\Rightarrow('s,'p,'f)\ com)\Rightarrow('a\Rightarrow's\ set)\Rightarrow('a\Rightarrow's\ set)\Rightarrow('s,'p,'f)\ com$
$\mathit{specAnno}\ P\ c\ Q\ A \equiv c\ \mathit{arbitrary}$

The assertions P, Q and A as well as the statement c depend on an auxiliary variable of polymorphic type *'a*. This auxiliary variable can be used to fix the state or to introduce logical variables. If we need more than one variable we can use a tuple. The statement c depends on the auxiliary variable, too. This enables nested annotations to refer to the auxiliary variable. After stripping all annotations the raw body should not refer to the variable. That is why the whole *specAnno* construct is defined as *c arbitrary*. The logical variable is only used by the verification condition generator. It has no semantical effect, not even a syntactic one with respect to the core language of Simpl. The Hoare rule for *specAnno* is mainly an instance of the consequence rule:

$$\frac{\begin{array}{c}P \subseteq \{s.\ \exists Z.\ s \in P'\,Z \wedge Q'\,Z \subseteq Q \wedge A'\,Z \subseteq A\} \\ \forall Z.\ \Gamma,\Theta\vdash_{/F} (P'\,Z)\ (c\ Z)\ (Q'\,Z),(A'\,Z) \qquad \forall Z.\ c\ Z = c\ \mathit{arbitrary}\end{array}}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{specAnno}\ P'\ c\ Q'\ A')\ Q,A}$$

The side-condition $\forall Z.\ c\ Z = c\ \mathit{arbitrary}$ expresses our intention about body c explained above: The raw body is independent of the auxiliary variable. This side-condition is solved automatically by the *vcg*. The concrete syntax for this specification annotation is shown in the following example. Consider we want to prove

$$\Gamma\vdash \{\!|\sigma.\ P\ \sigma|\!\}\ c_1;\ c_2;\ c_3\ \{\!|Q\ \sigma|\!\}.$$

The precondition $\{\!|\sigma.\ P\ \sigma|\!\}$ is an abbreviation for $\{s.\ s{=}\sigma \wedge P\ \sigma\}$. Hence the pre-state is fixed as σ, so that the postcondition can refer to the initial state. Now we can isolate statement c_2 and fix the state between c_1 and c_2 as τ:

$$\Gamma\vdash \{\!|\sigma.\ P\ \sigma|\!\}\ c_1;\ \mathit{ANNO}\ \tau.\ \{\!|\tau.\ P'\ \sigma\ \tau|\!\}\ c_2\ \{\!|Q'\ \sigma\ \tau|\!\};\ c_3\ \{\!|Q\ \sigma|\!\}.$$

The intermediate assertions can refer to both σ and τ. According to the rule for *specAnno* we can now prove the inner Hoare triple separately. The consequence side-condition ensures that the isolated Hoare triple fits into the main proof. The syntax hides that the nested c_2 is formally $\lambda\tau.\ c_2$. According to the definition of *specAnno* the whole inner triple $\mathit{ANNO}\ \tau.\ \{\!|\tau.\ P'\ \sigma\ \tau|\!\}\ c_2\ \{\!|Q'\ \sigma\ \tau|\!\}$ simplifies to c_2.

4.10 Conclusion

In this chapter I have presented the verification condition generator and the integration of the Hoare logic into Isabelle/HOL. This provides a solid verification environment for imperative programs. The examples in this chapter demonstrate that the verification condition generator results in quite natural proof obligations. With an additional modifies-clause we can lift separation of heap components, which are directly expressible in the split heap model, to the level of procedures, without having to introduce a new logic like separation logic [101]. Crucial parts of the frame problem can then already be handled during verification condition generation and the modifies-clause itself can be proven automatically.

The flexibility of the assertions as HOL sets can be exploited to deal with dynamic procedure calls and closures. An assertion in a Hoare triple can itself hold a nested Hoare triple to specify the behaviour of a procedure pointer or closure.

The examples in this chapter are all focused on the verification condition generator. However, the Hoare rules can also be applied by hand. Moreover, Wenzel [115] has shown how a Hoare logic can be integrated into an Isar proof. Here is an example of the verification of multiplication by iterated addition.

lemma
$\Gamma\vdash \{\!|\mathsf{m} = 0 \wedge \mathsf{s} = 0|\!\}$
WHILE $\mathsf{m} \neq a$

```
DO s := s + b; m := m + 1 OD
{|s = a * b|}
proof −
  let Γ⊢ - ?while - = ?thesis
  let {|?inv|} = {|s = m * b|}

  have {|m = 0 & s = 0|} ⊆ {|?inv|} by auto
  also have Γ⊢ … ?while {|?inv ∧ ¬ (m ≠ a)|}
  proof
    let ?c = s := s + b; m := m + 1
    have {|?inv ∧ m ≠ a|} ⊆ {|s + b = (m + 1) * b|}
      by auto
    also have Γ⊢ … ?c {|?inv|} by vcg
    finally show Γ⊢ {|?inv ∧ m ≠ a|} ?c {|?inv|} .
  qed
  also have {|?inv ∧ ¬ (m ≠ a)|} ⊆ {|s = a * b|} by auto
  finally show ?thesis by blast
qed
```

Without going into detail, let me highlight some aspects. With the **let** commands the term abbreviations *?while*, *?inv* and later *?c* are introduced by matching. The *?while* stores the whole loop statement, *?inv* matches the invariant and *?c* the loop body. The keyword **also** triggers transitivity reasoning. The set of transitivity rules is extensible. In case of the Hoare logic there are transitivity rules for consequence rules. For instance, the first **also** weakens the precondition to the invariant. The … abbreviate the last mentioned term. For instance, in the first case {|*?inv*|}. The nested proof is for the loop body. Here we use the verification condition generator to solve the preservation of the invariant.

The major parts of practical applications are proven by first invoking the verification condition generator. The resulting verification condition can still be proven by a structured Isar proof. However, the technique above can be used to decompose a Hoare triple into smaller pieces. This can be especially useful to isolate difficult sections of the program for some manual treatment.

CHAPTER **5**

Interfacing Program Analysis

This chapter explores how results of a program analysis or a software model checker can be introduced to the Hoare logic to support the verification.

Contents

Our Hoare logic based verification environment aims at the verification of functional correctness properties of programs. Hence the tool is essentially interactive. On the other hand, during the last years, advances in verification methodology as well as in computing power have resulted in tools for automatic program verification. Examples are Astrée [15, 66], Slam[1], Magic[2] and Blast[3]. These tools do not target full functional verification, but try to ensure safety properties like the absence of buffer overflows or dereferencing null pointers. In this section, we discuss how we can integrate such tools into our verification environment in order to increase automation and reduce the workload for the user. The focus is not on the technical integration but on the logical one. Which rules and concepts of the Hoare logic can be used as interface to the results produced by external tools?

In the Hoare logic for Simpl we distinguish three aspects of program verification:

- termination,
- absence of runtime faults and
- functional correctness.

[1]See also the Slam Home page for more information: `http://research.microsoft.com/slam/`
[2]See also the Magic Home page for more information: `http://www-2.cs.cmu.edu/~chaki/magic/`
[3]See also the Blast Home page for more information: `http://www-cad.eecs.berkeley.edu/~rupak/blast/`

The easiest tool to integrate is termination analysis. If termination is already proven, we can switch from total correctness to partial correctness. We can use the following Hoare rule to introduce the result to the Hoare logic:

$$\frac{\Gamma,\Theta\vdash_{/F} P\ c\ Q,A \qquad \forall s{\in}P.\ \Gamma\vdash c\downarrow \mathit{Normal}\ s}{\Gamma,\Theta\vdash_{t/F} P\ c\ Q,A}$$

The rule is applied backwards. A total correctness proof can be separated in partial correctness and termination. The termination tool has to supply the theorem $\forall s{\in}P.\ \Gamma\vdash c\downarrow \mathit{Normal}\ s$. Ideally it creates a proof of this property that can be checked by Isabelle. Isabelle also supplies a so called *oracle* interface. This way we can introduce a theorem as an axiom into the verification process without giving a proof for it. The results of the external tool have to be trusted if an oracle is used.

Runtime faults are modelled as explicit guards within Simpl. The basic idea is to discharge those guards that can be proven valid by an external tool. However, it is not always helpful for interactive verification if we just remove the guard from the program. The information of the guard can sometimes be valuable for functional correctness as well — for example in case of array bounds. Knowing that the index is within range is also important for reasoning about access operations. In this respect it would be counterproductive to remove the information that a program analysis has inferred. Then we have to reprove this constraint during the proof of functional correctness. Instead of removing the guard we want to be able to use it as granted for the further verification. The basic means to achieve this is already built into the Hoare logic. It is our notion of validity modulo faults in a set F which allows the rule:

$$\frac{f \in F \qquad \Gamma,\Theta\vdash_{/F} (g \cap P)\ c\ Q,A}{\Gamma,\Theta\vdash_{/F} P\ (\mathit{Guard}\ f\ g\ c)\ Q,A}$$

For the verification of the body c of the guarded statement we can assume the guard, without having to prove it, as long as the fault is in the set F.

Besides guarding against runtime faults, guards can also be utilised to integrate additional information that a program analysis has inferred, to support the functional correctness proof. The program analysis infers program properties for a certain program point. This information can be put into a guard that is considered to be valid.

As sketched above, the manipulation of guards plays a central role. Therefore, we introduce some auxiliary functions and predicates and important properties of them.

5.1 Stripping Guards

All guarded commands are marked with a fault f that is raised if the guard fails. The function *strip-guards* gets a set of faults F and removes all guards that are marked with a fault in this set.

◂ Definition 5.1

$$\begin{array}{ll}
\textit{strip-guards} :: \textit{'f set} \Rightarrow (\textit{'s,'p,'f})\ \textit{com} \Rightarrow (\textit{'s,'p,'f})\ \textit{com} & \\
\textit{strip-guards F Skip} & = \textit{Skip} \\
\textit{strip-guards F (Basic f)} & = \textit{Basic f} \\
\textit{strip-guards F (Spec r)} & = \textit{Spec r} \\
\textit{strip-guards F (Seq } c_1\ c_2) & = \textit{Seq (strip-guards F } c_1) (\textit{strip-guards F } c_2) \\
\textit{strip-guards F (Cond b } c_1\ c_2) & = \textit{Cond b (strip-guards F } c_1) (\textit{strip-guards F } c_2) \\
\textit{strip-guards F (While b c)} & = \textit{While b (strip-guards F c)} \\
\textit{strip-guards F (Call p)} & = \textit{Call p} \\
\textit{strip-guards F (DynCom } c_s) & = \textit{DynCom } (\lambda s.\ \textit{strip-guards F } (c_s\ s)) \\
\textit{strip-guards F (Guard f g c)} & = \textsf{if } f \in F \textsf{ then } \textit{strip-guards F c} \\
& \quad \textsf{else } \textit{Guard f g (strip-guards F c)} \\
\textit{strip-guards F Throw} & = \textit{Throw} \\
\textit{strip-guards F (Catch } c_1\ c_2) & = \textit{Catch (strip-guards F } c_1) (\textit{strip-guards F } c_2)
\end{array}$$

To strip the guards from the bodies stored in the procedure environment we introduce the function *strip*.

◂ Definition 5.2

$$\begin{array}{l}
\textit{strip} :: \textit{'f set} \Rightarrow (\textit{'p} \rightharpoonup (\textit{'s,'p,'f})\ \textit{com}) \Rightarrow (\textit{'p} \rightharpoonup (\textit{'s,'p,'f})\ \textit{com}) \\
\textit{strip F } \Gamma \equiv \lambda p.\ \textsf{case } \Gamma\ p \textsf{ of } \textit{None} \Rightarrow \textit{None} \mid \lfloor \textit{bdy} \rfloor \Rightarrow \lfloor \textit{strip-guards F bdy} \rfloor
\end{array}$$

If a guard is violated it raises the fault it is marked with. Otherwise a guarded command just executes its body. Hence if no fault occurs during execution the same run is possible for a program where the guards are stripped off. To be more precise, if no fault in *F* occurs during execution of statement *c*, then the same execution is possible for statement *strip-guards F c*.

◂ Lemma 5.1

If $\Gamma \vdash \langle c,s \rangle \Rightarrow t$ *and* $t \notin \textit{Fault}\ `\ F$ *then* $\Gamma \vdash \langle \textit{strip-guards F c}, s \rangle \Rightarrow t$.

Proof. By induction on the execution of *c*. □

We get the same result if we strip the context Γ.

◂ Lemma 5.2

If $\Gamma \vdash \langle c,s \rangle \Rightarrow t$ *and* $t \notin \textit{Fault}\ `\ F$ *then* $\textit{strip F }\Gamma \vdash \langle c,s \rangle \Rightarrow t$.

Proof. By induction on the execution in context Γ and Lemma 5.1. □

For the opposite direction the situation is a little more involved. If execution of *strip-guards F c* causes a runtime fault, then execution of *c* causes a runtime fault, too. However, it could be another fault, since *c* can contain some guards marked with a fault in *F*. If *c* causes a runtime fault not in *F* then it has to be the same. If *c* does not cause a runtime fault then the final states of both runs have to coincide.

◂ Lemma 5.3

$$\begin{array}{l}
\textit{If } \Gamma \vdash \langle \textit{strip-guards F c}, s \rangle \Rightarrow t \textit{ then} \\
\exists t'.\ \Gamma \vdash \langle c,s \rangle \Rightarrow t' \wedge \\
\quad (\textit{isFault } t \longrightarrow \textit{isFault } t') \wedge \\
\quad (t' \in \textit{Fault}\ `\ (- F) \longrightarrow t' = t) \wedge (\neg\ \textit{isFault } t' \longrightarrow t' = t).
\end{array}$$

Proof. By induction on the execution of *strip-guards F c*. □

◂ Lemma 5.4

$$\begin{array}{l}
\textit{If strip F } \Gamma \vdash \langle c,s \rangle \Rightarrow t \textit{ then} \\
\exists t'.\ \Gamma \vdash \langle c,s \rangle \Rightarrow t' \wedge \\
\quad (\textit{isFault } t \longrightarrow \textit{isFault } t') \wedge \\
\quad (t' \in \textit{Fault}\ `\ (- F) \longrightarrow t' = t) \wedge (\neg\ \textit{isFault } t' \longrightarrow t' = t).
\end{array}$$

Proof. By induction on the execution in context *strip F* Γ and Lemma 5.3. □

Causing a runtime fault is considered as termination in Simpl. Hence the more guards a program has the more likely it is to terminate.

Lemma 5.5 ► *If* $\Gamma\vdash \textit{strip-guards}\ F\ c \downarrow s$ *then* $\Gamma\vdash c \downarrow s$.

Proof. By induction on termination of *strip-guards c* and Lemma 5.1. □

Lemma 5.6 ► *If* $\textit{strip}\ F\ \Gamma\vdash c \downarrow s$ *then* $\Gamma\vdash c \downarrow s$.

Proof. By induction on termination in context *strip* Γ and Lemmas 5.5 and 5.2. □

For the other direction we have to exclude runtime faults in *F* for the execution of *c*.

Lemma 5.7 ► *If* $\Gamma\vdash c \downarrow \textit{Normal}\ s$ *and* $\Gamma\vdash \langle c,\textit{Normal}\ s\rangle \Rightarrow\notin \textit{Fault}\ `\ F$ *then* $\Gamma\vdash \textit{strip-guards}\ F\ c \downarrow \textit{Normal}\ s$.

Proof. By induction on termination of *c* and Lemma 5.3. □

Lemma 5.8 ► *If* $\Gamma\vdash c \downarrow \textit{Normal}\ s$ *and* $\Gamma\vdash \langle c,\textit{Normal}\ s\rangle \Rightarrow\notin \textit{Fault}\ `\ F$ *then* $\textit{strip}\ F\ \Gamma\vdash c \downarrow \textit{Normal}\ s$.

Proof. By induction on termination in context Γ and Lemmas 5.7 and 5.4. □

From the semantic properties of *strip-guards* we can derive the following rule:

$$\frac{\Gamma,\Theta\vdash_{/F} P\ c\ Q,A \qquad \Gamma,\Theta\vdash_{/\{\}}\ P\ (\textit{strip-guards}\ (-\ F)\ c)\ \textit{UNIV},\textit{UNIV}}{\Gamma,\Theta\vdash_{/\{\}}\ P\ c\ Q,A}$$

Again the intended application is backwards. The user wants to verify property $\Gamma,\Theta\vdash_{/\{\}}\ P\ c\ Q,A$. With the rule above, this goal can be reduced to $\Gamma,\Theta\vdash_{/F}\ P\ c\ Q,A$, where all the guards with a mark in *F* are granted. This is justified by the second premise, which ensures that no guard with a mark in *F* actually fails. We only leave those guards in *c* that are marked with *F* by applying *strip-guards* (− *F*) *c*. The only purpose of the second premise is to guarantee that the guards hold, hence the postconditions are trivial. The second premise is the one that is intended to be generated by the automatic tool.

For total correctness the rule is similar. In this case, partial correctness of the second premise suffices, since termination is handled by the first premise.

$$\frac{\Gamma,\Theta\vdash_{t/F} P\ c\ Q,A \qquad \Gamma,\Theta\vdash_{/\{\}}\ P\ (\textit{strip-guards}\ (-\ F)\ c)\ \textit{UNIV},\textit{UNIV}}{\Gamma,\Theta\vdash_{t/\{\}}\ P\ c\ Q,A}$$

To employ the rules we have to mark the guards according to the result of the automated tool.

5.2 Marking Guards

With *mark-guards f c* we mark all guards in statement *c* with fault *f*.

◄ Definition 5.3

$$
\begin{array}{ll}
\textit{mark-guards} :: {}'f \Rightarrow ({}'s,{}'p,{}'f)\ \textit{com} \Rightarrow ({}'s,{}'p,{}'f)\ \textit{com} & \\
\textit{mark-guards}\ f\ \textit{Skip} & = \textit{Skip} \\
\textit{mark-guards}\ f\ (\textit{Basic}\ g) & = \textit{Basic}\ g \\
\textit{mark-guards}\ f\ (\textit{Spec}\ r) & = \textit{Spec}\ r \\
\textit{mark-guards}\ f\ (\textit{Seq}\ c_1\ c_2) & = \textit{Seq}\ (\textit{mark-guards}\ f\ c_1)\ (\textit{mark-guards}\ f\ c_2) \\
\textit{mark-guards}\ f\ (\textit{Cond}\ b\ c_1\ c_2) & = \textit{Cond}\ b\ (\textit{mark-guards}\ f\ c_1)\ (\textit{mark-guards}\ f\ c_2) \\
\textit{mark-guards}\ f\ (\textit{While}\ b\ c) & = \textit{While}\ b\ (\textit{mark-guards}\ f\ c) \\
\textit{mark-guards}\ f\ (\textit{Call}\ p) & = \textit{Call}\ p \\
\textit{mark-guards}\ f\ (\textit{DynCom}\ c_s) & = \textit{DynCom}\ (\lambda s.\ \textit{mark-guards}\ f\ (c_s\ s)) \\
\textit{mark-guards}\ f\ (\textit{Guard}\ f'\ g\ c) & = \textit{Guard}\ f\ g\ (\textit{mark-guards}\ f\ c) \\
\textit{mark-guards}\ f\ \textit{Throw} & = \textit{Throw} \\
\textit{mark-guards}\ f\ (\textit{Catch}\ c_1\ c_2) & = \textit{Catch}\ (\textit{mark-guards}\ f\ c_1)\ (\textit{mark-guards}\ f\ c_2)
\end{array}
$$

Semantically, marking guards does not change much. It can only happen that the marked program causes the new fault instead of the old one.

◄ Lemma 5.9

If $\Gamma\vdash \langle c,s\rangle \Rightarrow t$ *then*
$\exists t'.\ \Gamma\vdash \langle \textit{mark-guards}\ f\ c,s\rangle \Rightarrow t' \wedge$
$\quad \textit{isFault}\ t = \textit{isFault}\ t' \wedge (\neg\ \textit{isFault}\ t \longrightarrow t' = t)$.

Proof. By induction on execution of statement *c*. □

And similarly in the other direction.

◄ Lemma 5.10

If $\Gamma\vdash \langle \textit{mark-guards}\ f\ c,s\rangle \Rightarrow t$ *then*
$\exists t'.\ \Gamma\vdash \langle c,s\rangle \Rightarrow t' \wedge$
$\quad \textit{isFault}\ t = \textit{isFault}\ t' \wedge$
$\quad (t' = \textit{Fault}\ f \longrightarrow t' = t) \wedge (\neg\ \textit{isFault}\ t' \longrightarrow t' = t)$.

Proof. By induction on execution of statement *mark-guards f c*. □

Marking guards has no effect on termination.

◄ Lemma 5.11

$\Gamma\vdash \textit{mark-guards}\ f\ c \downarrow s = \Gamma\vdash c \downarrow s$

Proof. In each direction by induction on the termination judgement and Lemmas 5.9 and 5.10. □

With these semantic properties of *mark-guards* it is evident that the following Hoare rules are valid:

$$
\frac{\Gamma,\Theta\vdash_{/\{\}}\ P\ c\ Q,A}{\Gamma,\Theta\vdash_{/\{\}}\ P\ (\textit{mark-guards}\ f\ c)\ Q,A}
\qquad
\frac{\Gamma,\Theta\vdash_{/\{\}}\ P\ (\textit{mark-guards}\ f\ c)\ Q,A}{\Gamma,\Theta\vdash_{/\{\}}\ P\ c\ Q,A}
$$

$$
\frac{\Gamma,\Theta\vdash_{t/\{\}}\ P\ c\ Q,A}{\Gamma,\Theta\vdash_{t/\{\}}\ P\ (\textit{mark-guards}\ f\ c)\ Q,A}
\qquad
\frac{\Gamma,\Theta\vdash_{t/\{\}}\ P\ (\textit{mark-guards}\ f\ c)\ Q,A}{\Gamma,\Theta\vdash_{t/\{\}}\ P\ c\ Q,A}
$$

5.3 Discharging Guards

Now we have all preliminaries to use the results of a program analysis or a software model checker to discharge some guards. We only have to distinguish between proven and unproven guards. Hence a Boolean mark is sufficient. The central rule is the following:

$$\frac{\begin{array}{cc}\Gamma,\Theta\vdash_{/\{True\}} P\ c'\ Q,A & \Gamma,\Theta\vdash_{/\{\}}\ P\ c''\ UNIV,UNIV \\ c = mark\text{-}guards\ False\ c' & c'' = strip\text{-}guards\ \{False\}\ c'\end{array}}{\Gamma,\Theta\vdash_{/\{\}}\ P\ c\ Q,A}\ (\textsc{DischargeGuards})$$

Initially we are in a state where we attempt to prove $\Gamma,\Theta\vdash_{/\{\}}\ P\ c\ Q,A$. We assume that all guards in c are marked as *False*. Note that the actual mark is not important since we currently work modulo the empty set of faults ($/\{\}$), which means that we have to prove all guards, regardless of their mark. The target statement c' is one that contains the same guards as c, but some may be marked as *True*. This is expressed by the equation $c = mark\text{-}guards\ False\ c'$. The first premise of the rule is the goal where the user continues, when all the other premises are discharged. This Hoare triple is annotated with $/\{True\}$ which means that all the guards in c' that are marked with *True* are considered as granted. The others still have to be proven by the user. The guards in c' that are marked with *True* have to be discharged by the second premise. The guards marked with *False* are stripped. This is expressed with equation $c'' = strip\text{-}guards\ \{False\}\ c'$. The premise $\Gamma,\Theta\vdash\ P\ c''\ UNIV,UNIV$ is the interface to the automatic tool. It describes the result of the program analysis in the terms of the Hoare logic. It ensures that all the guards in c'' hold, but nothing more, since the postcondition is trivial. The guards in c'' are exactly those guards of c' that are marked with *True*.

The rule for total correctness is analogous. In this case, again, partial correctness in the second premise suffices, since termination is handled by the first premise. This means that we do not require a termination proof from the program analysis.

$$\frac{\begin{array}{cc}\Gamma,\Theta\vdash_{t/\{True\}} P\ c'\ Q,A & \Gamma,\Theta\vdash_{/\{\}}\ P\ c''\ UNIV,UNIV \\ c = mark\text{-}guards\ False\ c' & c'' = strip\text{-}guards\ \{False\}\ c'\end{array}}{\Gamma,\Theta\vdash_{t/\{\}}\ P\ c\ Q,A}$$

The following example is a bubble sort implementation, with a number of guards to watch for array bound violations and arithmetic overflows. The list of guards in front of some statements is syntactic sugar for nested guarded commands. The default mark is *False*, which is omitted in the syntax. The current task for the user is to prove this Hoare triple:

```
Γ⊢ {|i = |arr| − 1 ∧ |arr| < max-nat|}
  WHILE 0 < i
  DO j := 0;
     WHILE j < i
     DO {|j + 1 ≤ max-nat|}, {|j + 1 < |arr||}, {|j < |arr||}
        ↦ IF arr[j + 1] < arr[j]
           THEN {|j < |arr||}↦ temp := arr[j];
            {|j < |arr||}, {|j + 1 ≤ max-nat|}, {|j + 1 < |arr||}
            ↦ arr[j] := arr[j + 1];
```

```
                $\{\!|j + 1 \leq \mathit{max\text{-}nat}|\!\}, \{\!|j + 1 < |arr||\!\} \mapsto arr_{[j\,+\,1]} := temp$
              FI;
          $\{\!|j + 1 \leq \mathit{max\text{-}nat}|\!\} \mapsto j := j + 1$
      OD;
      $\{\!|1 \leq i|\!\} \mapsto i := i - 1$
  OD
  $\{\!|\forall j{<}|arr|.\ \forall i{<}j.\ arr_{[i]} \leq arr_{[j]}|\!\}$
```

Now the user invokes the tactic to discharge some guards. The tactic passes the problem to the software model checker or the program analysis, which manages to solve some of the guards. The results are translated to the following Hoare triple[4]:

```
$\Gamma\vdash_{/\{\}}$ $\{\!|i = |arr| - 1 \wedge |arr| < \mathit{max\text{-}nat}|\!\}$
    WHILE $0 < i$
    DO $j := 0$;
        WHILE $j < i$
        DO $\{\!|j + 1 \leq \mathit{max\text{-}nat}|\!\}\surd, \{\!|j + 1 < |arr||\!\}\surd, \{\!|j < |arr||\!\}\surd$
            $\mapsto$ IF $arr_{[j\,+\,1]} < arr_{[j]}$
               THEN $\{\!|j < |arr||\!\}\surd \mapsto temp := arr_{[j]}$;
                 $arr_{[j]} := arr_{[j\,+\,1]}$;
                 $\{\!|j + 1 \leq \mathit{max\text{-}nat}|\!\}\surd, \{\!|j + 1 < |arr||\!\}\surd$
                 $\mapsto arr_{[j\,+\,1]} := temp$
               FI;
            $\{\!|j + 1 \leq \mathit{max\text{-}nat}|\!\}\surd \mapsto j := j + 1$
        OD;
        $\{\!|1 \leq i|\!\}\surd \mapsto i := i - 1$
    OD
    UNIV,UNIV
```

This Hoare triple only contains those guards that the software model checker could handle. They are marked with $\surd$ which is syntactic sugar for the mark *True*. For example, in the **THEN** branch there are some guards missing compared to the original statement. The tactic compares this result (c'') with the initial statement (c) and calculates the resulting statement (c') and instantiates the DischargeGuards Rule. The second side-condition is solved by the result of the software model checker and the other side-conditions by rewriting. All these steps are performed automatically. The next goal the user has to deal with stems from the first premise of the DischargeGuards Rule:

```
$\Gamma\vdash_{/\{True\}}$ $\{\!|i = |arr| - 1 \wedge |arr| < \mathit{max\text{-}nat}|\!\}$
        WHILE $0 < i$
        DO $j := 0$;
            WHILE $j < i$
            DO $\{\!|j + 1 \leq \mathit{max\text{-}nat}|\!\}\surd, \{\!|j + 1 < |arr||\!\}\surd, \{\!|j < |arr||\!\}\surd$
                $\mapsto$ IF $arr_{[j\,+\,1]} < arr_{[j]}$
                   THEN $\{\!|j < |arr||\!\}\surd \mapsto temp := arr_{[j]}$;
                     $\{\!|j < |arr||\!\}, \{\!|j + 1 \leq \mathit{max\text{-}nat}|\!\}, \{\!|j + 1 < |arr||\!\}$
                     $\mapsto arr_{[j]} := arr_{[j\,+\,1]}$;
                     $\{\!|j + 1 \leq \mathit{max\text{-}nat}|\!\}\surd, \{\!|j + 1 < |arr||\!\}\surd$
```

[4]The guards that are proven are arbitrarily chosen by me for this example. They do not reflect the current implementation of any software model checker or program analysis.

```
            ↦ arr[j + 1] := temp
          FI;
        ⦃j + 1 ≤ max-nat⦄√↦ j := j + 1
      OD;
      ⦃1 ≤ i⦄√↦ i := i − 1
    OD
    ⦃∀j<|arr|. ∀i<j. arr[i] ≤ arr[j]⦄
```

The mode has switched to $/\{True\}$ and the guards that were proven by the software model checker are ticked off, whereas the other guards are still marked with *False*. Now the user can continue by calling the verification condition generator. When the verification condition generator passes a ticked off guard it can either completely ignore it or use it as an assumption:

$$\frac{P \subseteq R \qquad \Gamma,\Theta \vdash_{/F} R\ c\ Q,A \qquad f \in F}{\Gamma,\Theta \vdash_{/F} P\ (\mathit{Guard}\ f\ g\ c)\ Q,A} \qquad \frac{P \subseteq \{s.\ s \in g \longrightarrow s \in R\} \qquad \Gamma,\Theta \vdash_{/F} R\ c\ Q,A \qquad f \in F}{\Gamma,\Theta \vdash_{/F} P\ (\mathit{Guard}\ f\ g\ c)\ Q,A}$$

By providing two syntactic variants for a ticked off guard the user can even specify the behaviour individually for each guard.

5.4 Adding Guards

The approach described so far is used to discharge guards that are already present in the original program. Those guards protect the program from runtime faults. However, program analysis may also be able to derive program properties beyond this class that can be used to verify (parts of) functional correctness. The idea is that the program analysis puts this information to additional guards in the program. Once these guards are added they can be discharged by the techniques described so far. The only missing piece is how to add guards to the program. We introduce the relation $c_1 \subseteq_g c_2$ that expresses that statement c_2 has more guards than c_1. Ignoring all guards, both c_1 and c_2 share the same skeleton.

Definition 5.4 ►

$$\begin{array}{ll}
\mathit{Skip} \subseteq_g \mathit{Skip} & = \mathit{True} \\
\mathit{Basic}\ f_1 \subseteq_g \mathit{Basic}\ f_2 & = f_1 = f_2 \\
\mathit{Spec}\ r_1 \subseteq_g \mathit{Spec}\ r_2 & = r_1 = r_2 \\
\mathit{Seq}\ c_1\ d_1 \subseteq_g \mathit{Seq}\ c_2\ d_2 & = (c_1 \subseteq_g c_2) \wedge (d_1 \subseteq_g d_2) \\
\mathit{Cond}\ b_1\ c_1\ d_1 \subseteq_g \mathit{Cond}\ b_2\ c_2\ d_2 & = b_1 = b_2 \wedge (c_1 \subseteq_g c_2) \wedge (d_1 \subseteq_g d_2) \\
\mathit{While}\ b_1\ c_1 \subseteq_g \mathit{While}\ b_2\ c_2 & = b_1 = b_2 \wedge (c_1 \subseteq_g c_2) \\
\mathit{Call}\ p_1 \subseteq_g \mathit{Call}\ p_2 & = p_1 = p_2 \\
\mathit{DynCom}\ c_1 \subseteq_g \mathit{DynCom}\ c_2 & = \forall s.\ c_1\ s \subseteq_g c_2\ s \\
\mathit{Guard}\ f_1\ g_1\ c_1 \subseteq_g \mathit{Guard}\ f_2\ g_2\ c_2 & = f_1 = f_2 \wedge g_1 = g_2 \wedge (c_1 \subseteq_g c_2) \vee \\
 & \quad (\mathit{Guard}\ f_1\ g_1\ c_1 \subseteq_g c_2) \\
c_1 \subseteq_g \mathit{Guard}\ f\ g\ c_2 & = c_1 \subseteq_g c_2 \\
\mathit{Throw} \subseteq_g \mathit{Throw} & = \mathit{True} \\
\mathit{Catch}\ c_1\ d_1 \subseteq_g \mathit{Catch}\ c_2\ d_2 & = (c_1 \subseteq_g c_2) \wedge (d_1 \subseteq_g d_2) \\
\text{-} \subseteq_g \text{-} & = \mathit{False}
\end{array}$$

A statement with more guards is more likely to cause a runtime fault. If it does not cause a runtime fault it yields the same result as the statement with fewer guards:

If $c \subseteq_g c'$ *and* $\Gamma\vdash \langle c,s\rangle \Rightarrow t$ *then*
$\exists t'.\ \Gamma\vdash \langle c',s\rangle \Rightarrow t' \wedge (\mathit{isFault}\ t \longrightarrow \mathit{isFault}\ t') \wedge (\neg\ \mathit{isFault}\ t' \longrightarrow t' = t)$. ◂ Lemma 5.12

Proof. By induction on c' and in case of the while loop a nested induction on the execution of c'. □

If a statement with more guards terminates and does not cause a runtime fault, then the statement with fewer guards terminates, too.

If $\Gamma\vdash c' \downarrow s$ *and* $c \subseteq_g c'$ *and* $\Gamma\vdash \langle c',s\rangle \Rightarrow\notin \mathit{Fault}\ `\ \mathit{UNIV}$ *then* $\Gamma\vdash c \downarrow s$. ◂ Lemma 5.13

Proof. By induction on c' and Lemma 5.12. In case of the while loop we use a nested induction on the termination of c'. □

These two lemmas justify the following Hoare rules. They allow us to introduce more guards to the statement.

$$\frac{c \subseteq_g c' \qquad \Gamma,\Theta\vdash_{/\{\}}\ P\ c'\ Q,A}{\Gamma,\Theta\vdash_{/\{\}}\ P\ c\ Q,A} \qquad \frac{c \subseteq_g c' \qquad \Gamma,\Theta\vdash_{t/\{\}}\ P\ c'\ Q,A}{\Gamma,\Theta\vdash_{t/\{\}}\ P\ c\ Q,A}$$

We can first use these rules to attach the results of a program analysis to the statement. Then we continue as described in the previous section to discharge these guards.

5.5 Conclusion

In this chapter I have presented an approach to integrate program analysis or software model checking into the process of functional verification within a Hoare logic based environment. The main characteristic of this approach is that it is declarative and modular. We do not have to instrument the verification condition generator to call the program analysis at certain program points. Instead, the program analysis can run before the verification condition generator and its result is clearly captured in a Hoare triple. This Hoare triple is then used to annotate the program with guards that can be taken as granted for the verification. Currently we have integrated a software model checker called ACSAR[5] and a termination analysis [94, 95, 24] in this fashion [26]. In the current state the software model checker discharges guards about arithmetic overflows and array bound checks. For instance, it is able to discharge all the guards in the bubble sort algorithm that was presented in this chapter. Currently the model checker is extended to handle null pointer checks.

Right now the tools do not produce any proof that can be checked by Isabelle. They are treated as trusted oracles. However, program analysis can be instrumented to produce a Hoare proof. This is proposed by Seo *et al.* [105] and independently by Chaieb [22] and implemented by Dehne [27] for an interval analysis. With such a proof we can close the gap and seamlessly integrate the program analysis into the verification environment without introducing any soundness issues.

I am not aware of any other work to integrate program analysis or software model checking in a Hoare logic based framework. However, a similar approach is

[5]See also the ACSAR Home page for more information `http://www.mpi-sb.mpg.de/~seghir/ACSAR/ACSAR-web-page.html`

followed by Wildmoser *et al.* [117] in the context of a proof carrying code framework, but on the level of byte-code.

The approach followed in context of the Why [33] tool is complementary to the one presented in this chapter. The Why tool can generate verification conditions for several theorem provers, among them, the automatic theorem prover Simplify[6] [75] and the interactive theorem prover Coq [14]. In some cases the verification condition can already be proven by Simplify, otherwise it has to be proven in Coq. In our setting a similar effect could be achieved by integrating Simplify as an oracle in Isabelle that is invoked on the verification condition. Such an integration is similar to the work on the connection of Isabelle to automatic first order theorem provers [69], SAT solvers [113] and arithmetic decision procedures [11]. In contrast the approach in this chapter integrates the results of the program analysis or software model checker *before* the verification condition generator is invoked. We do not expect the automatic tool to solve the complete verification condition, but can exploit the assertions it returns already during verification condition generation and also in the following interactive verification.

[6]See the Simplify home page: `http://research.compaq.com/SRC/esc/Simplify.html`

CHAPTER 6
Compositionality

In this Chapter, we examine how building and reusing verified libraries about imperative programs can be supported in the verification environment. The goal is to enable building verified libraries independent of each other and to supply means to combine these components to a program.

Contents

The state space of a Simpl program is modelled as a record. It depends on the program variables and the data structures that are stored in the heap. The type of the state space also affects the type of every Simpl statement: *('s, 'p, 'f) com*. The *'s* is instantiated with the state space type. Hence every Hoare triple is restricted to the state space type it was declared with. If the complete program is known, the state space can be fixed in advance, before the verification starts. Moreover the style of specifications that we have introduced in Sections 4.4 and 4.6 is robust with respect to extensions of the state space. The functional specifications and even the modifies-clause, only refer to the relevant parts of the state space in particular the heap. This means that after extending the state space all the specifications and proofs still work. However, we have to rerun the session.

6.1 Exploiting Structural Subtyping

We can avoid rerunning the session by exploiting the extensibility of records in Isabelle/HOL [74, 80]. Every record has an additional field "…" of a polymorphic type. For example:

record *one* =
 N :: *nat*

The record-type that is created has two fields $(\!|N :: nat, \ldots :: 'a|\!)$. This type is called *'a one-scheme*. Type *one* is an abbreviation for $(\!|N :: nat, \ldots :: unit|\!)$ or *unit one-scheme*, where *'a* is instantiated with the unit type. By instantiating *'a* with other types the record can be extended.

record *two* = *one* +
 M :: *nat*

This definition creates an extension type (|*M* :: *nat*, ... :: *'a*|) that contains the fields *M* :: *nat* and a new extension slot ... :: *'a*. By inserting (|*M* :: *nat*, ... :: *'a*|) in *'a one-scheme* we get (|*M* :: *nat*, ... :: *'a*|) *one-scheme*. This is (|*N* :: *nat*, *M* :: *nat*, ... :: *'a*|). Hence we have extended record *one* with the new fields of record *two*. The crucial point in this construction is, that record *two* is just an instance of record *one*. So every property that has been proven about a record with type *'a one-scheme* is also valid if we instantiate *'a* so that the resulting record is a *two* record. Linear extensions of records are possible with this kind of structural subtyping.

We can employ this kind of subtyping in the design of our state space. First, all the global components are grouped together in a single field. This allows us to generically copy all global components as in the return from a procedure. We have the following scheme for the state space:

record *'g state* =	**record** *globals* =
globals :: *'g*	... *<global variables + heap>* ...
... *<local variables>* ...	

As both the global components and the state are defined as record, they are both extensible. The *globals* component of record *state* is polymorphic, so that it works with any extension of record *globals*. With this setup we can achieve that global variables, heap as well as local variables can be extended. For example, we can start with a library of heap-lists and then extend it with heap-trees etc. The main benefit of this kind of compositionality is that it is for free. Mere type instantiation lifts the propositions to the extended state space. The drawback are the limitations of structural subtyping. We can only extend the state space in a linear fashion. We can start with the library for heap-lists and continue with heap-trees, or vice versa, but we cannot develop both theories independently in their minimal state space and then merge them. The only way to achieve this is to merge the state space in advance and then rerun the whole session. Moreover, we cannot reuse the same library in two instances. For example, a program may contain several data structures that are linked together as a list, like strings or queues. It would be nice if we could reuse the same verified list-reversal procedure for both instances. We are aiming at this kind of compositionality in the remainder of this chapter. We develop a framework that allows to lift a Hoare triple defined for a fixed state space into a Hoare triple for a bigger state space. Bigger means that the original state space can be embedded into the new one. We can project the original state out of a state from the bigger state space.

6.2 Lifting Hoare Triples

The relation between the original state space *'s* and the bigger one *'S* can be described by two functions:

$$\begin{array}{lll} \textit{project} & :: & 'S \Rightarrow 's \\ \textit{inject} & :: & 'S \Rightarrow 's \Rightarrow 'S \end{array}$$

Intuitively the original state space is a part of the bigger state space. With *project S* we can obtain the embedded original state space from *S*, and with *inject S s* we can replace the embedded original state in *S* by *s*. The functions *project* and *inject* are parameters of our framework that appear in the following definitions and theorems. We do not give any definitions for *project* and *inject* yet, but specify their expected behaviour axiomatically. Such common parameters and their specifications can be grouped together in Isabelle in a so called *locale* [10]. This allows us to develop the rest of the theory under these common assumptions. If we later on attempt to use the theory for a concrete example, we only have to instantiate the functions *project* and *inject* according to our needs and prove their specifications. All the theorems of the general theory are then automatically instantiated to the current application. The locale we define is named *lift-state-space*. To highlight which definitions and theorems depend on this locale they are marked with (in *lift-state-space*).

◄ Definition 6.1

locale *lift-state-space* =
fixes

$project :: 'S \Rightarrow 's$
$inject \ :: 'S \Rightarrow 's \Rightarrow 'S$

assumes

(1) Projection and injection commutes:
$project\ (inject\ S\ s) = s$

(2) Injection and projection commutes:
$inject\ S\ (project\ S) = S$

(3) Only the last injection matters:
$inject\ (inject\ S\ s)\ t = inject\ S\ t$

With the functions *project* and *inject* as basic building blocks we can define what it means to lift a command from one state space to another. We start with lifting of functions as they appear in the *Basic* command. The lifted function takes a state *S* of the new state space, applies the original function *f* to the projection of *S* and injects the result into *S*.

◄ Definition 6.2 (in *lift-state-space*)

$$lift_f :: ('s \Rightarrow 's) \Rightarrow ('S \Rightarrow 'S)$$
$$lift_f\ f \equiv \lambda S.\ inject\ S\ (f\ (project\ S))$$

Next, we lift state sets as they appear in the conditions of statements *Cond* and *While* or in assertions. A state belongs to the lifted set if the projected state belongs to the original set.

◄ Definition 6.3 (in *lift-state-space*)

$$lift_s :: 's\ set \Rightarrow 'S\ set$$
$$lift_s\ A \equiv \{S.\ project\ S \in A\}$$

The *Spec* command specifies the possible next state as a relation. Hence we need to lift relations, too.

◄ Definition 6.4 (in *lift-state-space*)

$$lift_r :: ('s \times 's)\ set \Rightarrow ('S \times 'S)\ set$$
$$lift_r\ r \equiv \{(S, T).\ (project\ S, project\ T) \in r \wedge T = inject\ S\ (project\ T)\}$$

A pair of states belongs to the lifted relation if the projections of the states belong to the original relation. This is the first part of the conjunction. The additional equation $T = inject\ S\ (project\ T)$ expresses that the "rest" of T is the same as S. In the *Spec* command the pair (S, T) describes a potential state transition from state S to state T. The effect of the embedded original state space is described by the relation r. The remaining parts of the lifted state space have to stay the same. This restriction is necessary to lift frame conditions or modifies-clauses (cf. Section 4.6). The original frame condition only expresses which parts of the projected state space stay the same. To enforce that the additional parts also remain unchanged we need this restriction.

Now we have all the ingredients to lift a command:

Definition 6.5 ► (in *lift-state-space*)

$$
\begin{array}{ll}
\multicolumn{2}{l}{lift_c :: ('s,'p,'f)\ com \Rightarrow ('S,'p,'f)\ com} \\
lift_c\ Skip & = Skip \\
lift_c\ (Basic\ f) & = Basic\ (lift_f\ f) \\
lift_c\ (Spec\ r) & = Spec\ (lift_r\ r) \\
lift_c\ (Seq\ c_1\ c_2) & = Seq\ (lift_c\ c_1)\ (lift_c\ c_2) \\
lift_c\ (Cond\ b\ c_1\ c_2) & = Cond\ (lift_s\ b)\ (lift_c\ c_1)\ (lift_c\ c_2) \\
lift_c\ (While\ b\ c) & = While\ (lift_s\ b)\ (lift_c\ c) \\
lift_c\ (Call\ p) & = Call\ p \\
lift_c\ (DynCom\ c_s) & = DynCom\ (\lambda s.\ lift_c\ (c_s\ (project\ s))) \\
lift_c\ (Guard\ f\ g\ c) & = Guard\ f\ (lift_s\ g)\ (lift_c\ c) \\
lift_c\ Throw & = Throw \\
lift_c\ (Catch\ c_1\ c_2) & = Catch\ (lift_c\ c_1)\ (lift_c\ c_2)
\end{array}
$$

Function $lift_e$ is used to lift the procedure environment:

Definition 6.6 ► (in *lift-state-space*)

$$
\begin{array}{l}
lift_e :: ('p \rightharpoonup ('s,'p,'f)\ com) \Rightarrow ('p \rightharpoonup ('S,'p,'f)\ com) \\
lift_e\ \Gamma\ p \equiv \textsf{case}\ \Gamma\ p\ \textsf{of}\ None \Rightarrow None \mid \lfloor bdy \rfloor \Rightarrow \lfloor lift_c\ bdy \rfloor
\end{array}
$$

Moreover, we define a state projection $project_x$ for extended states:

Definition 6.7 ► (in *lift-state-space*)

$$
\begin{array}{l}
project_x :: ('S,'f)\ xstate \Rightarrow ('s,'f)\ xstate \\
project_x\ s \equiv \\
\textsf{case}\ s\ \textsf{of}\ Normal\ s \Rightarrow Normal\ (project\ s) \mid Abrupt\ s \Rightarrow Abrupt\ (project\ s) \\
\mid Fault\ f \Rightarrow Fault\ f \mid Stuck \Rightarrow Stuck
\end{array}
$$

The original state space is embedded in the lifted state space. Hence an execution of the lifted program somehow contains the execution of the original program. This is proven in the following theorem:

Theorem 6.1 ► (in *lift-state-space*) *Simulation*

If $lift_e\ \Gamma \vdash \langle lift_c\ c,S\rangle \Rightarrow T$ *then* $\Gamma \vdash \langle c,project_x\ S\rangle \Rightarrow project_x\ T$.

Proof. By induction on the execution of the lifted statement. □

Theorem 6.1 is the key to lift partial correctness properties.

Theorem 6.2 ► (in *lift-state-space*) *Lift partial correctness*

If $\Gamma \models_{/F} P\ c\ Q,A$ *then* $lift_e\ \Gamma \models_{/F} (lift_s\ P)\ (lift_c\ c)\ (lift_s\ Q),(lift_s\ A)$.

Proof. According to Definition 3.2 of validity we have to consider an execution $lift_e\ \Gamma \vdash \langle lift_c\ c,Normal\ S\rangle \Rightarrow T$ of the lifted program, where $S \in lift_s\ P$ and $T \notin Fault\ `\ F$. We have to show $T \in Normal\ `\ lift_s\ Q \cup Abrupt\ `\ lift_s\ A$. From the simulation Theorem 6.1 we get $\Gamma \vdash \langle c,project_x\ (Normal\ S)\rangle \Rightarrow project_x\ T$, and as S satisfies the lifted precondition we also have *project* $S \in P$. Since we know the validity of $\Gamma\models_{/F} P\ c\ Q,A$, we know that the projected final state satisfies the postcondition: $\mathit{project}_x\ T \in \mathit{Normal}\ `\ Q \cup \mathit{Abrupt}\ `\ A$. With Definitions 6.3 and 6.7 this is lifted to $T \in \mathit{Normal}\ `\ \mathit{lift}_s\ Q \cup \mathit{Abrupt}\ `\ \mathit{lift}_s\ A$. □

With the soundness and completeness Theorems 3.8 and 3.12 this result can be converted into a Hoare rule:

$$\frac{\Gamma\vdash_{/F} P\ c\ Q,A}{\mathit{lift}_e\ \Gamma\vdash_{/F} (\mathit{lift}_s\ P)\ (\mathit{lift}_c\ c)\ (\mathit{lift}_s\ Q),(\mathit{lift}_s\ A)}\ (\textsc{Lift})$$

To lift total correctness properties we also have to lift termination.

◂ Theorem 6.3 (in *lift-state-space*) *Lift termination*

If $\Gamma\vdash c \downarrow \mathit{project}_x\ S$ *then* $\mathit{lift}_e\ \Gamma\vdash \mathit{lift}_c\ c \downarrow S$.

Proof. By induction on the termination of *c* and Theorem 6.1. □

Together with Theorem 6.2 we can now lift total correctness properties.

◂ Theorem 6.4 (in *lift-state-space*) *Lift total correctness*

If $\Gamma\models_{t/F} P\ c\ Q,A$ *then* $\mathit{lift}_e\ \Gamma\models_{t/F} (\mathit{lift}_s\ P)\ (\mathit{lift}_c\ c)\ (\mathit{lift}_s\ Q),(\mathit{lift}_s\ A)$.

With the soundness and completeness Theorems 3.15 and 3.28 this result can be converted to a Hoare rule:

$$\frac{\Gamma\vdash_{t/F} P\ c\ Q,A}{\mathit{lift}_e\ \Gamma\vdash_{t/F} (\mathit{lift}_s\ P)\ (\mathit{lift}_c\ c)\ (\mathit{lift}_s\ Q),(\mathit{lift}_s\ A)}\ (\textsc{Lift})$$

There is a subtle effect on the properties we can lift, which stems from the definition of lift_s. A lifted assertion only describes a property of the embedded original state. For “usual” assertions this is exactly what we want to achieve, but for frame conditions we want more. A frame condition is a property of the whole state space not only of some part that is changed by the program. Those parts of the lifted state that are not part of the embedded state remain unchanged during execution. We want to include this information to the frame condition as we lift it.

The function *state* takes an extended state and yields the raw state in case it is *Normal* or *Abrupt*.

◂ Definition 6.8

$$\begin{aligned} &\mathit{state} :: ('s,'f)\ \mathit{xstate} \Rightarrow 's\\ &\mathit{state}\ (\mathit{Normal}\ s) = s\\ &\mathit{state}\ (\mathit{Abrupt}\ s)\ = s \end{aligned}$$

The following lemma expresses that during the execution of a lifted statement only the embedded state changes. We exclude *Fault* and *Stuck* final states, since only *Abrupt* or *Normal* states have a sensible raw state that is obtained by function *state*. Moreover, *Fault* and *Stuck* final states are already covered by Theorem 6.1.

◂ Lemma 6.5 (in *lift-state-space*)

If $\mathit{lift}_e\ \Gamma\vdash \langle \mathit{lift}_c\ c,S\rangle \Rightarrow T$ *and* $T \notin \mathit{Fault}\ `\ \mathit{UNIV} \cup \{\mathit{Stuck}\}$ *then*
$\mathit{state}\ T = \mathit{inject}\ (\mathit{state}\ S)\ (\mathit{project}\ (\mathit{state}\ T))$.

Proof. By induction on the execution of the lifted statement. □

With this lemma we can properly lift frame conditions. The first conjunct in the following postconditions takes care of the embedded state, whereas the second conjunct takes care of the additional parts.

Theorem 6.6 ► (in *lift-state-space*) *Lift frame condition*

If $\forall s.\ \Gamma\models_{/F} \{s\}\ c\ (Modif\ s),(ModifAbr\ s)$ *then*
$lift_e\ \Gamma\models_{/F}$
$\{S\}\ (lift_c\ c)$
$\{T.\ T \in lift_s\ (Modif\ (project\ S)) \wedge T = inject\ S\ (project\ T)\}$,
$\{T.\ T \in lift_s\ (ModifAbr\ (project\ S)) \wedge T = inject\ S\ (project\ T)\}$.

Proof. Analogous to Theorem 6.2 using Lemma 6.5. □

The rules we have introduced allow all kinds of manipulations on programs, like renaming of variables or heap components or merging two libraries. To merge two libraries that are defined for different state spaces we first have to define a new state space that is capable of storing all the components of the libraries state spaces. Then we define the *project* and *inject* functions for both libraries. Note that the requirements of locale *lift-state-space* on the *project* and *inject* functions allow us to share parts of the state between the two libraries. This can be used to merge two libraries about heap lists, for example. However, right now we cannot rename procedures. This is necessary if we want to reuse a procedure twice for different data structures. For example, a list library may be used to implement strings as well as queues. To provide this possibility we introduce the function *rename* to rename the procedure calls in a statement according to a name mapping *N*.

Definition 6.9 ► (in *lift-state-space*)

$$\begin{array}{ll}
\multicolumn{2}{l}{rename :: ('p \Rightarrow 'q) \Rightarrow ('s,'p,'f)\ com \Rightarrow ('s,'q,'f)\ com} \\
rename\ N\ Skip & = Skip \\
rename\ N\ (Basic\ f) & = Basic\ f \\
rename\ N\ (Spec\ r) & = Spec\ r \\
rename\ N\ (Seq\ c_1\ c_2) & = Seq\ (rename\ N\ c_1)\ (rename\ N\ c_2) \\
rename\ N\ (Cond\ b\ c_1\ c_2) & = Cond\ b\ (rename\ N\ c_1)\ (rename\ N\ c_2) \\
rename\ N\ (While\ b\ c) & = While\ b\ (rename\ N\ c) \\
rename\ N\ (Call\ p) & = Call\ (N\ p) \\
rename\ N\ (DynCom\ c_s) & = DynCom\ (\lambda s.\ rename\ N\ (c_s\ s)) \\
rename\ N\ (Guard\ f\ g\ c) & = Guard\ f\ g\ (rename\ N\ c) \\
rename\ N\ Throw & = Throw \\
rename\ N\ (Catch\ c_1\ c_2) & = Catch\ (rename\ N\ c_1)\ (rename\ N\ c_2)
\end{array}$$

We also have to rename the procedure environment Γ. We require that all defined procedures are stored in Γ′ at the renamed position:

$$\forall p\ bdy.\ \Gamma\ p = \lfloor bdy\rfloor \longrightarrow \Gamma'\ (N\ p) = \lfloor rename\ N\ bdy\rfloor$$

We make no assumptions about the undefined procedures in Γ. The new environment Γ′ can define more procedures as Γ. It can happen that renaming defines a previously undefined procedure. Then the execution of the original program may end up in a *Stuck* state, whereas the renamed program might not. Since the Hoare logic excludes *Stuck* final states anyway this special case imposes no problems.

Theorem 6.7 ► *If* $\forall p\ bdy.\ \Gamma\ p = \lfloor bdy\rfloor \longrightarrow \Gamma'\ (N\ p) = \lfloor rename\ N\ bdy\rfloor$ *and* $\Gamma'\vdash \langle rename\ N\ c,s\rangle \Rightarrow t$ *then* $\exists t'.\ \Gamma\vdash \langle c,s\rangle \Rightarrow t' \wedge (t' = Stuck \vee t' = t)$.

Proof. By induction on the execution of the renamed program. □

Theorem 6.8 ► *If* $\forall p\ bdy.\ \Gamma\ p = \lfloor bdy\rfloor \longrightarrow \Gamma'\ (N\ p) = \lfloor rename\ N\ bdy\rfloor$ *and* $\Gamma\models_{/F} P\ c\ Q,A$ *then* $\Gamma'\models_{/F} P\ (rename\ N\ c)\ Q,A$.

Proof. By Theorem 6.7 and Definition 3.2. □

To transfer termination to the renamed program we also have to exclude *Stuck* final states.

If $\forall p\ bdy.\ \Gamma\ p = \lfloor bdy \rfloor \longrightarrow \Gamma'\ (N\ p) = \lfloor rename\ N\ bdy \rfloor$ *and* $\Gamma \vdash c \downarrow s$ *and* $\Gamma \vdash \langle c,s \rangle \Rightarrow \notin \{Stuck\}$ *then* $\Gamma' \vdash rename\ N\ c \downarrow s$ ◂ Theorem 6.9

Proof. By induction on the termination of c and Theorem 6.7. □

Hence we can also transfer total correctness properties.

If $\forall p\ bdy.\ \Gamma\ p = \lfloor bdy \rfloor \longrightarrow \Gamma'\ (N\ p) = \lfloor rename\ N\ bdy \rfloor$ *and* $\Gamma \models_{t/F} P\ c\ Q,A$ *then* $\Gamma' \models_{t/F} P\ (rename\ N\ c)\ Q,A$. ◂ Theorem 6.10

Proof. By Theorems 6.8 and 6.9 and Definition 3.8. □

6.3 Example

As an example, we lift a general list reversal procedure to work with both strings and queues. We begin with the definition of the library that contains the list reversal procedure. The *next* heap is the only global component. The local variables are *p*, *q* and *r*.

record *state-list* =
globals :: *globals-list*
p :: *ref*
q :: *ref*
r :: *ref*

record *globals-list* =
next :: *ref* ⇒ *ref*

We define the list reversal procedure in this state space.

```
procedures Rev (p|q) =
q := NULL;
WHILE p ≠ NULL
DO r := p; {|p ≠ NULL|}↦ p := p→next;
   {|r ≠ NULL|}↦ r→next := q; q := r
OD
```

As described in Section 4.4 this definition creates a constant *Rev-body* for the body of the procedure and a locale *Rev-impl* which holds the single assumption: Γ *"Rev"* = ⌊*Rev-body*⌋. When we prove the specification and the frame condition for the list reversal these theorems are under the assumptions of locale *Rev-impl*. This means that they hold for any procedure environment Γ for which the list reversal is defined, in particular for the minimal environment Γ = [*"Rev"* ↦ *Rev-body*]. Here is the specifiction of *Rev*:

$$\forall Ps.\ \Gamma \vdash \{\!|List\ \mathsf{p}\ \mathsf{next}\ Ps|\!\}\ \mathsf{q} := \textbf{PROC}\ Rev(\mathsf{p})\ \{\!|List\ \mathsf{q}\ \mathsf{next}\ (rev\ Ps)|\!\}$$ ◂ Lemma 6.11 (in Rev-impl)

And here the frame condition.

Lemma 6.12 ► (in Rev-impl) $\forall \sigma.\ \Gamma\vdash_{/UNIV} \{\sigma\}$ q := **PROC** *Rev*(p) $\{t.\ t$ *may-only-modify-globals* σ *in* $[next]\}$

Next we define the extended state space. We want to import the list reversal twice, for strings and for queues.

```
struct string {                struct queue {
  char chr;                      int cont;
  struct string* strnext;        struct queue* qnext;
}                              }
```

The layout of the new state space is the following:

record *state* =	**record** *globals* =
globals :: *globals*	*chr* :: *ref* ⇒ *char*
str :: *ref*	*strnext* :: *ref* ⇒ *ref*
queue :: *ref*	
q :: *ref*	*cont* :: *ref* ⇒ *int*
r :: *ref*	*qnext* :: *ref* ⇒ *ref*

For strings we map the heap *strnext* to *next* and for queues the heap *qnext*. Similarly, the local variable *str* is mapped to *p* for strings and *queue* is mapped to *p* for queues. For *q* and *r* we share the local variables. To make use of our framework we have to define the *project* and *inject* functions. We start with strings. The projection function takes a state *S* of type *state* and yields the embedded state of type *state-list*.

project-globals-str:: *globals* ⇒ *globals-list*
project-globals-str G ≡ ⦇*next* = *strnext G*⦈

project-str:: *state* ⇒ *state-list*
project-str S ≡
⦇*state-list.globals* = *project-globals-str* (*globals S*),
state-list.p = *str S*, *state-list.q* = *q S*, *state-list.r* = *r S*⦈

The injection function takes a state *S* of type *state* and updates it according to the components in state *s* of type *state-list*.

inject-globals-str:: *globals* ⇒ *globals-list* ⇒ *globals*
inject-globals-str G g ≡ *G*⦇*strnext* := *next g*⦈

inject-str:: *state* ⇒ *state-list* ⇒ *state*
inject-str S s ≡
S⦇*globals* := *inject-globals-str* (*globals S*) (*globals-list.globals s*),
str := *state-list.p s*, *q* := *state-list.q s*,*r* := *state-list.r s*⦈

For these definitions the assumption of locale *lift-state-space* (cf. Definition 6.1) hold, which is proven automatically by Isabelle's simplifier. Hence we can use the lifting rule from Theorem 6.2. We lift the specification of the list reversal for the minimal procedure environment.

Lemma 6.13 ► $\forall Ps.\ lift_e\ [''Rev'' \mapsto Rev\text{-}body]\vdash$ ⦃*List* str strnext *Ps*⦄ *Call ''Rev''* ⦃*List* q strnext (*rev Ps*)⦄

Now we also rename the procedure to *RevStr*. We define the name mapping $\mathcal{N}$:

$$\mathcal{N} :: string \Rightarrow string$$
$$\mathcal{N}\ p \equiv \text{if } p = \text{"}Rev\text{" then "}RevStr\text{" else ""}$$

Then we define the new procedure as lifted version of *Rev*:

procedures $RevStr(str|q) = rename\ \mathcal{N}\ (lift_c\ Rev\text{-}body)$

For the procedure environment Γ in locale *RevStr-impl* we can prove the premise of Theorem 6.8. All procedures defined in the lifted minimal environment are also defined in Γ:

$$\forall p\ bdy.\ lift_e\ [\text{"}Rev\text{"} \mapsto Rev\text{-}body]\ p = \lfloor bdy \rfloor \longrightarrow \Gamma\ (\mathcal{N}\ p) = \lfloor rename\ \mathcal{N}\ bdy \rfloor$$

◄ Lemma 6.14 (in RevStr-impl)

Hence we can finally lift Lemma 6.13 to the new environment.

$$\forall Ps.\ \Gamma \vdash \{\!|List\ \text{str strnext}\ Ps|\!\}\ \text{q := } \mathbf{PROC}\ RevStr(\text{str})\ \{\!|List\ \text{q strnext}\ (rev\ Ps)|\!\}$$

◄ Lemma 6.15 (in RevStr-impl)

Analogously we lift the frame condition with the rule derived from Theorem 6.6.

$$\forall \sigma.\ \Gamma \vdash_{/UNIV} \{\sigma\}\ \text{q := } \mathbf{PROC}\ RevStr(\text{str})\ \{t.\ t\ may\text{-}only\text{-}modify\text{-}globals\ \sigma\ in\ [strnext]\}$$

◄ Lemma 6.16 (in RevStr-impl)

For the queue we can do exactly the same. We define the projection and injection functions from the queue components to the state space of the generic list reversal library and define a proper renaming function.

For the list reversal the only relevant heap component is the *next* pointer. The other fields like *chr* or *cont* are not necessary to lift the list reversal procedure. Hence every data-structure that somehow is a linked list can import the list reversal procedure, regardless of any additional component. An interesting question is how generic libraries can become if they also refer to the content of the list, for example, if we want to sort the list according to its content. We can use the type polymorphism of HOL in order to define the content generically. We assume that there is a field *content* of type *'a*. Then we can specify the procedure generically with respect to an ordering on type *'a* that is also a parameter of the specification. To instantiate the library we have to map the actual content to the generic *content* field. In our example this is *chr* for the strings or *cont* for the queues. If the content contains more than one field, this is also no problem. We can simply map the tuple of all the fields to *'a*. The *project* and *inject* functions can deal with this more general kind of embedding.

6.4 Conclusion

The presented framework provides a flexible foundation for the construction of verified libraries. It allows to build modules independently of each other and merge them into a program. The lifting of Hoare triples and the renaming of procedures works schematically and can thus be automated. Although the work was motivated by the state space representation as a record the theory is independent of this specific model and can hence be used in other set-ups as well.

The approach is independent of the restrictions of the embedded programming language. For example in C there is no notion of genericity, at least no type-safe one. Nevertheless, it is possible to develop and verify a generic list theory in the verification environment. Every instance of a list that appears in a given C program can then be derived from this generic list theory. This relieves the user from copying and reproving several instances of the same algorithms and data structures. This scenario actually appeared in the Verisoft Project.

Moreover, it seems promising to integrate this work with the refinement framework of Tuch and Klein [109]. They start with an abstract view of the system and prove the crucial properties on this level. The system is defined semantically as an abstract data type that consists of a set of initial states and a set of operations that specify possible state transitions of the system. Then they refine the model until a C implementation is reached. The meta theory of data refinement ensures that the properties for the abstract system are preserved by the refinement step, as long as each of the operations is implemented correctly. As the C level is reached the Hoare logic is used to prove the correctness of the individual operations. Furthermore, as Simpl provides the *Spec* command to specify a command, rather than to implement it, Simpl programs themselves can be refined. The framework of this chapter can be used to simplify the integration of various abstract data types into a complete system.

CHAPTER **7**

The C0 Language

This chapter introduces C0, a type-safe subset of the C programming language. The abstract syntax and the semantics of C0 is defined and a type system and a definite assignment analysis is developed. Type soundness theorems relate these static semantics to execution. In the following chapter, C0 is embedded into Simpl. Properties of C0 programs can be proven in the verification environment for Simpl.

Contents

C0 is a subset of the C programming language [57]. In this chapter, we develop a formal model of the C0 language and in the following chapter, we embed it into Simpl in order to employ the verification environment to derive program properties. We prove the soundness of this embedding. This means that the program properties proven in the verification environment are also valid in the C0 model.

The C0 programming language is used throughout the Verisoft project aiming at the pervasive verification of computer systems comprising hard and software. A complete system stack is developed and verified, containing a processor, a micro kernel, a simple operating system (including a TCP/IP stack) and a distributed application and a C0 compiler to the processor. C0 is used throughout all the software layers. To cover all abstraction levels various models of C0 were developed. The most abstract one is a big-step semantics. To reason about concurrency also a small-step semantics was developed. Moreover there is a variant of the C0 small-step semantics that allows to include in-line assembler instructions. This is necessary to describe parts of the micro kernel. The embedding of C0 into Simpl provides an even more abstract layer on top of the big-step semantics. To achieve high productivity due to a high level of abstraction, the goal is to cover as much of the system verification on the Simpl level as possible. However, since not all parts (like the in-line assembler code) are covered by the Simpl level we need a way to transfer the properties proven on the abstract level to the more concrete one. The soundness proof in the next chapter is one of those transformation steps. Another one is a simulation theorem of the C0 small-step semantics within the C0 big-step semantics. The key differences between the C0 big-step semantics and Simpl are:

- a monolithic heap in C0 versus a split heap model in Simpl and
- a deep embedding of expressions in C0 versus shallow embedding in Simpl.

This comparison reflects the different purposes of the language models. The C0 semantics is used to verify properties of the programming language, like type soundness or compiler correctness, whereas Simpl aims at the verification of individual programs.

The main motivation of *C0* was to identify a subset of C that is easy to verify but can still be compiled by an ordinary C compiler. That way one can use the standard C infrastructure to develop the programs. All aspects of C that might complicate verification and which can be avoided in the implementation work within Verisoft are omitted. The main features of C that were dropped are:

- `goto`s,
- abrupt termination (`break`, `continue` or `return`),
- side-effects in expressions,
- pointer arithmetic,
- unions and
- pointer casts.

So in the end C0 is semantically more like Pascal, but with C syntax. For some examples I use C syntax, since the corresponding C0 (abstract) syntax is less readable. The C0 model I present builds on the work of Martin Strecker for the Verisoft project. The main aspect I added is the definite assignment analysis (cf. Section 7.2.7) and on top of it the refined notion of state conformance and type safety (cf. Section 7.2.9).

7.1 Abstract Syntax

7.1.1 Names

To improve readability we introduce four (HOL) types for C0 identifiers: *tname* (type names), *vname* (variable names), *fname* (field names) and *pname* (procedure names). All these names are synonyms for *string*.

7.1.2 Types

C0 is a typed language.

◂ Definition 7.1 *C0 types*

C0 types are defined as a recursive datatype ty with the following constructors:

- *Boolean,*
- *Integer for signed integers,*
- *UnsgndT for unsigned intergers,*
- *CharT ,*
- *Ptr tn, where tn* :: *tname,*
- *NullT,*
- *Arr n T, where n* :: *nat, T* :: *ty, and*
- *Struct fs, where fs* :: (*fname* × *ty*) *list.*

In C0 the size of an array is already statically fixed by its type. The amount of memory occupied for each value of a certain type is determined by the function *sizeof-type*:

◂ Definition 7.2 *sizeof-type*

$$
\begin{array}{lcl}
\textit{sizeof-type} :: \textit{ty} \Rightarrow \textit{nat} & & \\
\textit{sizeof-type Boolean} & = & 1 \\
\textit{sizeof-type Integer} & = & 1 \\
\textit{sizeof-type UnsgndT} & = & 1 \\
\textit{sizeof-type CharT} & = & 1 \\
\textit{sizeof-type} \ (\textit{Ptr tn}) & = & 1 \\
\textit{sizeof-type NullT} & = & 1 \\
\textit{sizeof-type} \ (\textit{Arr n T}) & = & n * \textit{sizeof-type T} \\
\textit{sizeof-type} \ (\textit{Struct fTs}) & = & \textit{foldl} \ (+) \ 0 \ (\textit{map sizeof-type} \ (\textit{map snd fTs}))
\end{array}
$$

All types, except for structures and arrays are regarded as primitive types.

◂ Definition 7.3 *prim-type*

$$
\begin{array}{l}
\textit{primitive-type} :: \textit{ty} \Rightarrow \textit{bool} \\
\textit{prim-type T} \equiv \mathsf{case}\ T\ \mathsf{of}\ \textit{Struct fs} \Rightarrow \textit{False} \mid \textit{Arr n T} \Rightarrow \textit{False} \mid \text{-} \Rightarrow \textit{True}
\end{array}
$$

The numeric types are *Integer*, *UnsgndT* and *CharT*.

◂ Definition 7.4 *numeric-type*

$$
\begin{array}{l}
\textit{numeric-type} :: \textit{ty} \Rightarrow \textit{bool} \\
\textit{numeric-type T} \equiv T \in \{\textit{Integer}, \textit{UnsgndT}, \textit{CharT}\}
\end{array}
$$

7.1.3 Values

Definition 7.5 ► *C0 primitive values*

Primitive C0 values of (HOL) type prim can be:

- *a Boolean Bool b, where b* :: *bool,*
- *a (signed) integer Intg i, where i* :: *int,*
- *an unsigned integer Unsgnd n, where n* :: *nat,*
- *a character Chr c, where c* :: *int,*
- *a reference Addr a, where a* :: *loc* (cf. Definition 7.8)*, or*
- *the null reference Null.*

Definition 7.6 ► *C0 values*

C0 values of (HOL) type val can be:

- *primitive values Prim p, where p* :: *prim,*
- *arrays Arrv vs, where vs* :: *val list, or*
- *structures Structv fs, where fs* :: (*fname* × *val*) *list.*

We also define a set of destructors *the-*. . . :

Definition 7.7 ► *Value destructors*

$$
\begin{array}{lcl}
\textit{the-Prim}\ (\textit{Prim}\ v) & = & v \\
\textit{the-Bool}\ (\textit{Bool}\ b) & = & b \\
\textit{the-Bool}_v\ (\textit{Prim}\ (\textit{Bool}\ b)) & = & b \\
\textit{the-Intg}\ (\textit{Intg}\ i) & = & i \\
\textit{the-Intg}_v\ (\textit{Prim}\ (\textit{Intg}\ i)) & = & i \\
\textit{the-Unsgnd}\ (\textit{Unsgnd}\ n) & = & n \\
\textit{the-Unsgnd}_v\ (\textit{Prim}\ (\textit{Unsgnd}\ n)) & = & n \\
\textit{the-Chr}\ (\textit{Chr}\ c) & = & c \\
\textit{the-Chr}_v\ (\textit{Prim}\ (\textit{Chr}\ c)) & = & c \\
\textit{the-Addr}\ (\textit{Addr}\ a) & = & a \\
\textit{the-Addr}_v\ (\textit{Prim}\ (\textit{Addr}\ a)) & = & a \\
\textit{the-Structv}\ (\textit{Structv}\ \textit{fs}) & = & \textit{fs} \\
\textit{the-Arrv}\ (\textit{Arrv}\ \textit{vs}) & = & \textit{vs}
\end{array}
$$

The address model for the C0 big-step semantics is rather abstract. No assumptions about data-alignment or consecutive addresses are made. One location can store any kind of value, even structured ones. This is quite similar to references in Simpl as introduced in Section 2.4.9. For a convenient translation of C0-addresses to Simpl we identify the type *loc* of locations with the non *NULL* references *ref* by an Isabelle type definition:

Definition 7.8 ► *Locations*

$$\textbf{typedef}\ \textit{loc} = \{r.\ r \neq \textit{NULL}\}$$

Since there is no extra layer of values in Simpl, *NULL* is an ordinary element of type *ref*, whereas in C0 *Null* is an extra constructor of values. The definition of type *loc* allows us to map value *Null* to reference *NULL* since it can not be occupied by a C0 address. The functions *Ref* and *the-Ref* convert between *ref* and *val*. They use the functions *Abs-loc* and *Rep-loc* that are generated by the type definition facility of Isabelle to convert between *ref* and *loc*.

◂ Definition 7.9

$Ref :: ref \Rightarrow val$
$Ref\ r \equiv \textsf{if}\ r = NULL\ \textsf{then}\ Prim\ Null\ \textsf{else}\ Prim\ (Addr\ (Abs\text{-}loc\ r))$

◂ Definition 7.10

$$\begin{array}{lcl} \multicolumn{3}{l}{the\text{-}Ref :: val \Rightarrow ref} \\ the\text{-}Ref\ (Prim\ Null) & = & NULL \\ the\text{-}Ref\ (Prim\ (Addr\ l)) & = & Rep\text{-}loc\ l \end{array}$$

Heap memory in C0 is initialised upon allocation. The auxiliary function *default-val* yields the default value for every C0 type. Most importantly, every pointer is initialised with *Null*. Hence C0 avoids the issue of dangling pointers since it does not support deallocation of memory by the program. It is supposed to run with a garbage collector.

◂ Definition 7.11
default-val

$$\begin{array}{lcl} \multicolumn{3}{l}{default\text{-}val :: ty \Rightarrow val} \\ default\text{-}val\ Boolean & = & Prim\ (Bool\ False) \\ default\text{-}val\ Integer & = & Prim\ (Intg\ 0) \\ default\text{-}val\ UnsgndT & = & Prim\ (Unsgnd\ 0) \\ default\text{-}val\ CharT & = & Prim\ (Chr\ 0) \\ default\text{-}val\ (Ptr\ tn) & = & Prim\ Null \\ default\text{-}val\ NullT & = & Prim\ Null \\ default\text{-}val\ (Arr\ n\ T) & = & Arrv\ (replicate\ n\ (default\text{-}val\ T)) \\ default\text{-}val\ (Struct\ fs) & = & Structv\ (map\ (\lambda(n, T).\ (n, default\text{-}val\ T))\ fs) \end{array}$$

7.1.4 Expressions

Every expression carries a polymorphic tag *'a* that is supposed to store the type of the expression. Initially a C0 expression only carries a dummy tag (). A type elaboration phase annotates the (sub-)expressions with their types. For the purpose of this work we are only concerned with type annotated expressions. In the sequel we therefore use variable *T* for these type tags.

◂ Definition 7.12
C0 expressions

C0 supports the following expressions of (HOL) datatype 'a expr:

- *literal values Lit v T, where v :: val,*
- *variable access VarAcc vn T, where vn :: vname,*
- *array access ArrAcc e i T of array e with index i where e, i :: 'a expr,*
- *structure access StructAcc e n T of structure e with field n where e :: 'a expr and n :: fname,*
- *dereferencing Deref e T, where e :: 'a expr,*
- *unary operations UnOp uop e T, where uop :: unop and e :: 'a expr. Type unop comprises the following alternatives: unary-minus, bitwise-neg, logical-not and the casts to-int, to-unsigned-int and to-char,*
- *binary operations BinOp bop e_1 e_2 T, where bop :: binop and e_1, e_2 :: 'a expr. Type binop comprises the following alternatives: plus, minus, times, divide, bitwise-or, bitwise-and, bitwise-xor, shiftleft, shiftright, greater, less, equal, greaterequal, lessequal or notequal, and*

- *lazy binary operations $LazyBinOp\ bop\ e_1\ e_2\ T$, where bop :: $lazybinop$ and e_1, e_2 :: $'a\ expr$. A $lazybinop$ can either be logical-and or logical-or.*

With the selector *typ* we can get hold of the type tag of an expression:

Definition 7.13 ►

$$\begin{array}{lcl}
typ :: 'a\ expr \Rightarrow 'a & & \\
typ\ (Lit\ v\ T) & = & T \\
typ\ (VarAcc\ vn\ T) & = & T \\
typ\ (ArrAcc\ e\ i\ T) & = & T \\
typ\ (StructAcc\ e\ cn\ T) & = & T \\
typ\ (BinOp\ bop\ e_1\ e_2\ T) & = & T \\
typ\ (LazyBinOp\ bop\ e_1\ e_2\ T) & = & T \\
typ\ (UnOp\ uop\ e\ T) & = & T \\
typ\ (Deref\ e\ T) & = & T
\end{array}$$

7.1.5 Statements

The type variable *'a* for the (type-) tags of expressions is propagated to statements.

Definition 7.14 ►
C0 statements

C0 supports the following statements of (HOL) datatype 'a stmt:

- *the empty statement Skip,*
- *assignment of expression e_2 to (left-) expressions e_1: $Ass\ e_1\ e_2$, where both e_1 and e_2 are of type 'a expr,*
- *pointer allocation of a new element of type tn and assignment of the address to (left-) expression e: PAlloc e tn, where e :: 'a expr and tn :: tname,*
- *sequential composition $Comp\ c_1\ c_2$, where c_1, c_2 :: 'a stmt,*
- *conditional execution $Ifte\ e\ c_1\ c_2$, where e :: 'a expr and c_1, c_2 :: 'a stmt,*
- *while loop Loop e c, where e :: 'a expr and c :: 'a stmt,*
- *procedure/function call SCall vn pn ps, of procedure pn with parameters ps and return value assigned to variable vn, where vn :: vname, pn :: pname and ps :: 'a expr list, and*
- *return from procedure/function Return e.*

Procedures in C0 are statements. A procedure call can only occur as an assignment to a variable. They are of the form `vn = pn(ps)` in C syntax. Procedure calls can not appear in expressions. Hence expressions are free of side-effects.

7.1.6 Programs

A type declaration (*tdecl*) consists of a type name and its type. A variable declaration (*vdecl*) consists of a variable name and its type. A procedure declaration *pdecl* consists of the parameter declarations, the local variable declarations and the return type. A procedure definition (*'a pdefn*) consists of the procedure name, the procedure declaration and the body statement. Finally a program (*'a prog*) consists of type, global variable and procedure declarations.

◂ Definition 7.15
C0 program

$$\begin{array}{lcl} tdecl & = & tname \times ty \\ vdecl & = & vname \times ty \\ pdecl & = & vdecl\ list \times vdecl\ list \times ty \\ 'a\ pdefn & = & pname \times pdecl \times 'a\ stmt \\ 'a\ prog & = & tdecl\ list \times vdecl\ list \times 'a\ pdefn\ list \end{array}$$

◂ Definition 7.16

To extract declaration information of procedures and programs, we define various auxiliary functions. The definitions are self explanatory:

- *Components of procedure declarations:*
  $pardecls\text{-}of :: pdecl \Rightarrow vdecl\ list$
  $pardecls\text{-}of \equiv fst$

  $locdecls\text{-}of :: pdecl \Rightarrow vdecl\ list$
  $locdecls\text{-}of \equiv fst \circ snd$

  $returnT\text{-}of :: pdecl \Rightarrow ty$
  $returnT\text{-}of \equiv snd \circ snd$

- *Components of procedure definitions:*
  $pdecl\text{-}of :: (pdecl \times 'a\ stmt) \Rightarrow pdecl$
  $pdecl\text{-}of \equiv fst$

  $pbody\text{-}of :: (pdecl \times 'a\ stmt) \Rightarrow 'a\ stmt$
  $pbody\text{-}of \equiv snd$

- *Components of programs:*
  $tdecls\text{-}of :: 'a\ prog \Rightarrow tdecl\ list$
  $tdecls\text{-}of \equiv fst$

  $gdecls\text{-}of :: 'a\ prog \Rightarrow vdecl\ list$
  $gdecls\text{-}of \equiv fst \circ snd$

  $pdefns\text{-}of :: 'a\ prog \Rightarrow 'a\ pdefn\ list$
  $pdefns\text{-}of \equiv snd \circ snd$

- *Type, global variable and procedure lookup:*
  $tnenv :: 'a\ prog \Rightarrow (tname \rightharpoonup ty)$
  $tnenv \equiv map\text{-}of \circ tdecls\text{-}of$

  $genv :: 'a\ prog \Rightarrow (vname \rightharpoonup ty)$
  $genv \equiv map\text{-}of \circ gdecls\text{-}of$

  $plookup :: 'a\ prog \Rightarrow (pname \rightharpoonup (pdecl \times 'a\ stmt))$
  $plookup \equiv map\text{-}of \circ pdefns\text{-}of$

7.2 Semantics

7.2.1 State

The state of a C0 execution consists of a heap (mapping from locations to values), the local and global variables (mapping from variable names to values) and a counter for the free heap.

Definition 7.17 ► *C0 state*

record *state* =
  heap :: *loc* ⇀ *val*
  lvars :: *vname* ⇀ *val*
  gvars :: *vname* ⇀ *val*
  free-heap :: *nat*

Local variables may hide global variables. Throughout execution we keep track of the local variables via a set L to decide whether a name refers to a local or global variable. Lookup and update of variables depends on this set L:

Definition 7.18 ► *Variable lookup/update*

$$\begin{array}{ll} \textit{lookup-var} :: \textit{vname set} \Rightarrow \textit{state} \Rightarrow \textit{vname} \rightharpoonup \textit{val} & \\ \textit{lookup-var } L\ s\ vn & \equiv \textsf{if } vn \in L \textsf{ then } \textit{lvars } s\ vn \textsf{ else } \textit{gvars } s\ vn \end{array}$$

$$\begin{array}{ll} \textit{update-var} :: \textit{vname set} \Rightarrow \textit{state} \Rightarrow \textit{vname} \Rightarrow \textit{val} \Rightarrow \textit{state} & \\ \textit{update-var } L\ s\ vn\ v & \equiv \textsf{if } vn \in L \textsf{ then } s(\!|\textit{lvars} := \textit{lvars } s(vn \mapsto v)|\!) \\ & \quad \textsf{else } s(\!|\textit{gvars} := \textit{gvars } s(vn \mapsto v)|\!) \end{array}$$

7.2.2 Expression Evaluation

Expressions are evaluated to values. This evaluation may cause runtime faults like array bound violations or dereferencing null pointers. We model this behaviour with optional values. If evaluation succeeds a result of the form $\lfloor v \rfloor$ is obtained, otherwise *None*.

Definition 7.19 ► *Expression evaluation*

Expression evaluation: eval L s e, of expression e in state s and in context of local variables L is defined in Figure 7.1.

Expression evaluation is straightforward. The expression is processed recursively. If evaluation of a sub-expression fails this is propagated to the top.

The evaluation of unary and binary operations is mapped to the corresponding HOL definitions by the auxiliary functions *apply-unop* (cf. Figure 7.4), *apply-binop* (cf. Figure 7.3) and *apply-lazybinop* (cf. Figure 7.2). For lazy binary operations the auxiliary function *early-result* is used to decide whether the overall result is already determined by the first sub-expression. In this case evaluation of the second sub-expression is skipped. Since expressions do not have side-effects this only amounts to runtime errors. If the second expression is skipped it cannot cause a runtime error. Hence the first expression can implement a test to guard the second expression. A prominent idiom in C is that the first part of a conjunction implements a null pointer test for the following dereferenced expression. For example: `if p != null && p->x < 5 ....`

Definition 7.20 ►

$$\begin{array}{l} \textit{early-result} :: \textit{lazybinop} \Rightarrow \textit{bool} \Rightarrow \textit{bool option} \\ \textit{early-result bop } b \equiv \\ \textsf{case } \textit{bop} \textsf{ of } \textit{logical-and} \Rightarrow \textsf{if } b \textsf{ then } \textit{None} \textsf{ else } \lfloor b \rfloor \\ \mid \textit{logical-or} \Rightarrow \textsf{if } b \textsf{ then } \lfloor b \rfloor \textsf{ else } \textit{None} \end{array}$$

```
$eval :: vname\ set \Rightarrow state \Rightarrow 'a\ expr \Rightarrow val\ option$

$eval\ L\ s\ (Lit\ v\ T) = \lfloor v \rfloor$

$eval\ L\ s\ (VarAcc\ vn\ T) = lookup\text{-}var\ L\ s\ vn$

$eval\ L\ s\ (ArrAcc\ e\ i\ T) =$
case $eval\ L\ s\ e$ of $None \Rightarrow None$
| $\lfloor ev \rfloor \Rightarrow$
  case $eval\ L\ s\ i$ of $None \Rightarrow None$
  | $\lfloor iv \rfloor \Rightarrow$
    let $a = the\text{-}Arrv\ ev;\ n = the\text{-}Unsgnd_v\ iv$
    in if $n < |a|$ then $\lfloor a_{[n]} \rfloor$ else $None$

$eval\ L\ s\ (StructAcc\ e\ cn\ T) =$
case $eval\ L\ s\ e$ of $None \Rightarrow None$ | $\lfloor v \rfloor \Rightarrow map\text{-}of\ (the\text{-}Structv\ v)\ cn$

$eval\ L\ s\ (Deref\ e\ T) =$
case $eval\ L\ s\ e$ of $None \Rightarrow None$
| $\lfloor v \rfloor \Rightarrow$ if $v = Prim\ Null$ then $None$ else $heap\ s\ (the\text{-}Addr_v\ v)$

$eval\ L\ s\ (UnOp\ uop\ e\ T) =$
case $eval\ L\ s\ e$ of $None \Rightarrow None$
| $\lfloor v \rfloor \Rightarrow option\text{-}map\ Prim\ (apply\text{-}unop\ (uop,\ the\text{-}Prim\ v))$

$eval\ L\ s\ (BinOp\ bop\ e_1\ e_2\ T) =$
case $eval\ L\ s\ e_1$ of $None \Rightarrow None$
| $\lfloor v_1 \rfloor \Rightarrow$
  case $eval\ L\ s\ e_2$ of $None \Rightarrow None$
  | $\lfloor v_2 \rfloor \Rightarrow option\text{-}map\ Prim\ (apply\text{-}binop\ (bop,\ the\text{-}Prim\ v_1,\ the\text{-}Prim\ v_2))$

$eval\ L\ s\ (LazyBinOp\ lbop\ e_1\ e_2\ T) =$
case $eval\ L\ s\ e_1$ of $None \Rightarrow None$
| $\lfloor v_1 \rfloor \Rightarrow$ case $early\text{-}result\ lbop\ (the\text{-}Bool_v\ v_1)$ of
          $None \Rightarrow$ case $eval\ L\ s\ e_2$ of $None \Rightarrow None$
                    | $\lfloor v_2 \rfloor \Rightarrow option\text{-}map\ Prim\ (apply\text{-}lazybinop\ (lbop,\ the\text{-}Prim\ v_1,\ the\text{-}Prim\ v_2))$
          | $\lfloor b \rfloor \Rightarrow \lfloor Prim\ (Bool\ b) \rfloor$
```

Figure 7.1: Evaluation of C0 expressions

For the sequel the exact definition of operations like bit-shifting is not important and hence it is omitted. The only relevant point about the unary and binary operations is that C0 supports modulo arithmetic with silent over- and underflows. The default HOL modulo operation always yields a positive result, e.g. $-2\ mod\ 4 = 2$. To properly handle signed modulo arithmetic of C0 we define a variant *modwrap* where $-2\ modwrap\ 4 = -2$:

$$a\ modwrap\ m \equiv (a + m)\ mod\ (2 * m) - m$$ ◂ Definition 7.21

Function *modwrap* shifts the result of *mod* so that it lies between $-\ m$ (inclusively) and m (exclusively), for a positive m.

$$
\begin{array}{l}
\textit{apply-lazybinop} :: (\textit{lazybinop} \times \textit{prim} \times \textit{prim}) \Rightarrow \textit{prim option} \\[4pt]
\textit{apply-lazybinop}\ (\textit{logical-and}, \textit{Bool}\ b_1, \textit{Bool}\ b_2) = \lfloor \textit{Bool}\ (b_1 \wedge b_2) \rfloor \\
\textit{apply-lazybinop}\ (\textit{logical-and}, \textit{Bool False}, v) = \lfloor \textit{Bool False} \rfloor \\[4pt]
\textit{apply-lazybinop}\ (\textit{logical-or}, \textit{Bool}\ b_1, \textit{Bool}\ b_2) = \lfloor \textit{Bool}\ (b_1 \vee b_2) \rfloor \\
\textit{apply-lazybinop}\ (\textit{logical-or}, \textit{Bool True}, v) = \lfloor \textit{Bool True} \rfloor \\[4pt]
\textit{apply-lazybinop}\ (\text{-},\text{-},\text{-}) = \textit{None}
\end{array}
$$

Figure 7.2: Evaluation of lazy binary operations

The bounds, word and bit-sizes of the numeric types are listed in the following table.

constant	value
wlen-byte	*4*
bytelen-bit	*8*
wlen-bit	*wlen-byte ∗ bytelen-bit*
int-lb	*− int-ub*
int-ub	*2 ^ (wlen-bit − 1)*
un-int-ub	*2 ^ wlen-bit*
chr-lb	*− chr-ub*
chr-ub	*2 ^ (bytelen-bit − 1)*

We call a primitive value *bounded* if it respects the bounds of its type:

Definition 7.22 ►

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textit{bounded} :: \textit{prim} \Rightarrow \textit{bool}} \\
\textit{bounded}\ (\textit{Intg}\ i) & = & \textit{int-lb} \le i \wedge i < \textit{int-ub} \\
\textit{bounded}\ (\textit{Unsgnd}\ n) & = & n < \textit{un-int-ub} \\
\textit{bounded}\ (\textit{Chr}\ i) & = & \textit{chr-lb} \le i \wedge i < \textit{chr-ub} \\
\textit{bounded}\ \text{-} & = & \textit{True}
\end{array}
$$

To evaluate the parameters of a procedure call we lift evaluation of expressions to expression lists:

Definition 7.23 ►

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textit{evals} :: \textit{vname set} \Rightarrow \textit{state} \Rightarrow \textit{'a expr list} \Rightarrow \textit{val list option}} \\
\textit{evals}\ L\ s\ [] & = & \lfloor [] \rfloor \\
\textit{evals}\ L\ s\ (e{\cdot}es) & = & \mathsf{case}\ \textit{eval}\ L\ s\ e\ \mathsf{of}\ \textit{None} \Rightarrow \textit{None} \\
 & & \mid \lfloor v \rfloor \Rightarrow \\
 & & \quad \mathsf{case}\ \textit{evals}\ L\ s\ es\ \mathsf{of}\ \textit{None} \Rightarrow \textit{None} \\
 & & \quad \mid \lfloor vs \rfloor \Rightarrow \lfloor v{\cdot}vs \rfloor
\end{array}
$$

The big-step semantics propagates runtime faults, too. Therefore the program states of type *state* are embedded into the option type. Function *eval-opt* lifts expression evaluation to these optional states:

Definition 7.24 ►

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textit{eval-opt} :: \textit{vname set} \Rightarrow \textit{state option} \Rightarrow \textit{'a expr} \Rightarrow \textit{val option}} \\
\textit{eval-opt}\ L\ \textit{None}\ e & = & \textit{None} \\
\textit{eval-opt}\ L\ \lfloor s \rfloor\ e & = & \textit{eval}\ L\ s\ e
\end{array}
$$

$\mathit{apply\text{-}binop} :: (\mathit{binop} \times \mathit{prim} \times \mathit{prim}) \Rightarrow \mathit{prim\ option}$

$\mathit{apply\text{-}binop}\ (\mathit{equal}, v_1, v_2) = \lfloor \mathit{Bool}\ (v_1 = v_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{notequal}, v_1, v_2) = \lfloor \mathit{Bool}\ (v_1 \neq v_2) \rfloor$

$\mathit{apply\text{-}binop}\ (\mathit{plus}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) = \lfloor \mathit{Intg}\ ((i_1 + i_2)\ \mathit{modwrap}\ \mathit{int\text{-}ub}) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{plus}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Unsgnd}\ ((n_1 + n_2)\ \mathit{mod}\ \mathit{un\text{-}int\text{-}ub}) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{plus}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) = \lfloor \mathit{Chr}\ ((i_1 + i_2)\ \mathit{modwrap}\ \mathit{chr\text{-}ub}) \rfloor$

$\mathit{apply\text{-}binop}\ (\mathit{minus}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) = \lfloor \mathit{Intg}\ ((i_1 - i_2)\ \mathit{modwrap}\ \mathit{int\text{-}ub}) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{minus}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Unsgnd}\ (\mathit{nat}\ ((\mathit{int}\ n_1 - \mathit{int}\ n_2)\ \mathit{mod}\ \mathit{int}\ \mathit{un\text{-}int\text{-}ub})) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{minus}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) = \lfloor \mathit{Chr}\ ((i_1 - i_2)\ \mathit{modwrap}\ \mathit{chr\text{-}ub}) \rfloor$

$\mathit{apply\text{-}binop}\ (\mathit{times}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) = \lfloor \mathit{Intg}\ (i_1 * i_2\ \mathit{modwrap}\ \mathit{int\text{-}ub}) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{times}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Unsgnd}\ (n_1 * n_2\ \mathit{mod}\ \mathit{un\text{-}int\text{-}ub}) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{times}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) = \lfloor \mathit{Chr}\ (i_1 * i_2\ \mathit{modwrap}\ \mathit{chr\text{-}ub}) \rfloor$

$\mathit{apply\text{-}binop}\ (\mathit{divides}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) = \textsf{if}\ i_2 = 0\ \textsf{then}\ \mathit{None}\ \textsf{else}\ \lfloor \mathit{Intg}\ (i_1\ \mathit{div}\ i_2\ \mathit{modwrap}\ \mathit{int\text{-}ub}) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{divides}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) = \textsf{if}\ n_2 = 0\ \textsf{then}\ \mathit{None}\ \textsf{else}\ \lfloor \mathit{Unsgnd}\ (n_1\ \mathit{div}\ n_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{divides}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) = \textsf{if}\ i_2 = 0\ \textsf{then}\ \mathit{None}\ \textsf{else}\ \lfloor \mathit{Chr}\ (i_1\ \mathit{div}\ i_2\ \mathit{modwrap}\ \mathit{chr\text{-}ub}) \rfloor$

$\mathit{apply\text{-}binop}\ (\mathit{bitwise\text{-}or}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) = \lfloor \mathit{Intg}\ (i_1 \vee_s i_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{bitwise\text{-}or}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Unsgnd}\ (n_1 \vee_u n_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{bitwise\text{-}or}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) = \lfloor \mathit{Chr}\ (i_1 \vee_s i_2) \rfloor$

$\mathit{apply\text{-}binop}\ (\mathit{bitwise\text{-}and}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) = \lfloor \mathit{Intg}\ (i_1 \wedge_s i_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{bitwise\text{-}and}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Unsgnd}\ (n_1 \wedge_u n_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{bitwise\text{-}and}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) = \lfloor \mathit{Chr}\ (i_1 \wedge_s i_2) \rfloor$

$\mathit{apply\text{-}binop}\ (\mathit{bitwise\text{-}xor}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) = \lfloor \mathit{Intg}\ (i_1 \oplus_s i_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{bitwise\text{-}xor}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Unsgnd}\ (n_1 \oplus_u n_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{bitwise\text{-}xor}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) = \lfloor \mathit{Chr}\ (i_1 \oplus_s i_2) \rfloor$

$\mathit{apply\text{-}binop}\ (\mathit{shiftleft}, \mathit{Intg}\ i_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Intg}\ (i_1 \ll_{s/\mathit{wlen\text{-}bit}} n_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{shiftleft}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Unsgnd}\ (n_1 \ll_{u/\mathit{wlen\text{-}bit}} n_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{shiftleft}, \mathit{Chr}\ i_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Chr}\ (i_1 \ll_{s/\mathit{bytelen\text{-}bit}} n_2) \rfloor$

$\mathit{apply\text{-}binop}\ (\mathit{shiftright}, \mathit{Intg}\ i_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Intg}\ (i_1 \gg_{s/\mathit{wlen\text{-}bit}} n_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{shiftright}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Unsgnd}\ (n_1 \gg_{u/\mathit{wlen\text{-}bit}} n_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{shiftright}, \mathit{Chr}\ i_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Chr}\ (i_1 \gg_{s/\mathit{bytelen\text{-}bit}} n_2) \rfloor$

$\mathit{apply\text{-}binop}\ (\mathit{greater}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) = \lfloor \mathit{Bool}\ (i_2 < i_1) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{greater}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Bool}\ (n_2 < n_1) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{greater}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) = \lfloor \mathit{Bool}\ (i_2 < i_1) \rfloor$

$\mathit{apply\text{-}binop}\ (\mathit{less}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) = \lfloor \mathit{Bool}\ (i_1 < i_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{less}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Bool}\ (n_1 < n_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{less}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) = \lfloor \mathit{Bool}\ (i_1 < i_2) \rfloor$

$\mathit{apply\text{-}binop}\ (\mathit{greaterequal}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) = \lfloor \mathit{Bool}\ (i_2 \leq i_1) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{greaterequal}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Bool}\ (n_2 \leq n_1) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{greaterequal}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) = \lfloor \mathit{Bool}\ (i_2 \leq i_1) \rfloor$

$\mathit{apply\text{-}binop}\ (\mathit{lessequal}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) = \lfloor \mathit{Bool}\ (i_1 \leq i_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{lessequal}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) = \lfloor \mathit{Bool}\ (n_1 \leq n_2) \rfloor$
$\mathit{apply\text{-}binop}\ (\mathit{lessequal}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) = \lfloor \mathit{Bool}\ (i_1 \leq i_2) \rfloor$

$\mathit{apply\text{-}binop}\ (\text{-},\text{-},\text{-}) = \mathit{None}$

Figure 7.3: Evaluation of binary operations

$$
\begin{array}{l}
\textit{apply-unop} :: (\textit{unop} \times \textit{prim}) \Rightarrow \textit{prim option} \\[1ex]
\textit{apply-unop}\ (\textit{unary-minus}, \textit{Intg}\ i) = \lfloor \textit{Intg}\ (-\ i\ \textit{modwrap int-ub}) \rfloor \\
\textit{apply-unop}\ (\textit{unary-minus}, \textit{Chr}\ i) = \lfloor \textit{Chr}\ (-\ i\ \textit{modwrap chr-ub}) \rfloor \\[1ex]
\textit{apply-unop}\ (\textit{bitwise-neg}, \textit{Intg}\ i) = \lfloor \textit{Intg}\ \neg_{s/\textit{wlen-bit}}\ i \rfloor \\
\textit{apply-unop}\ (\textit{bitwise-neg}, \textit{Unsgnd}\ n) = \lfloor \textit{Unsgnd}\ \neg_{u/\textit{wlen-bit}}\ n \rfloor \\
\textit{apply-unop}\ (\textit{bitwise-neg}, \textit{Chr}\ i) = \lfloor \textit{Chr}\ \neg_{s/\textit{bytelen-bit}}\ i \rfloor \\[1ex]
\textit{apply-unop}\ (\textit{logical-not}, \textit{Bool}\ b) = \lfloor \textit{Bool}\ (\neg\ b) \rfloor \\[1ex]
\textit{apply-unop}\ (\textit{to-int}, \textit{Intg}\ i) = \lfloor \textit{Intg}\ i \rfloor \\
\textit{apply-unop}\ (\textit{to-int}, \textit{Unsgnd}\ n) = \lfloor \textit{Intg}\ (\textit{int}\ n\ \textit{modwrap int-ub}) \rfloor \\
\textit{apply-unop}\ (\textit{to-int}, \textit{Chr}\ i) = \lfloor \textit{Intg}\ i \rfloor \\[1ex]
\textit{apply-unop}\ (\textit{to-unsigned-int}, \textit{Intg}\ i) = \lfloor \textit{Unsgnd}\ (\textit{nat}\ (i\ \textit{mod int-ub})) \rfloor \\
\textit{apply-unop}\ (\textit{to-unsigned-int}, \textit{Unsgnd}\ n) = \lfloor \textit{Unsgnd}\ n \rfloor \\
\textit{apply-unop}\ (\textit{to-unsigned-int}, \textit{Chr}\ i) = \lfloor \textit{Unsgnd}\ (\textit{nat}\ (i\ \textit{mod chr-ub})) \rfloor \\[1ex]
\textit{apply-unop}\ (\textit{to-char}, \textit{Intg}\ i) = \lfloor \textit{Chr}\ (i\ \textit{modwrap chr-ub}) \rfloor \\
\textit{apply-unop}\ (\textit{to-char}, \textit{Unsgnd}\ n) = \lfloor \textit{Chr}\ (\textit{int}\ n\ \textit{modwrap chr-ub}) \rfloor \\
\textit{apply-unop}\ (\textit{to-char}, \textit{Chr}\ i) = \lfloor \textit{Chr}\ i \rfloor \\[1ex]
\textit{apply-unnop}\ (\text{-},\text{-}) = \textit{None}
\end{array}
$$

Figure 7.4: Evaluation of unary operations

7.2.3 Left-Expression Evaluation and Assignment

The left hand side of a C0 assignment is not restricted to plain variables but can contain an arbitrarily nested combination of variable-, structure- and array-access and dereferencing pointers. Following the C nomenclature, these left hand sides of assignments are called *left-expressions*. They evaluate to *left-values*. On real hardware such a left-value corresponds to a memory address. However, the C0 memory model we have introduced is not fine grained enough. Even structured values fit in one memory cell. A memory location can not address a sub-component of a structure or array. To keep this convenient level of abstraction we instead introduce a kind of path to address a sub-component of a compound value. We use the (left-)expressions themselves to describe this path. A left-value is a reduced left-expression, where all array indexes and addresses to dereference a pointer are evaluated. For example, term *Deref* (*Lit* (*Prim* (*Addr a*)) *T′*) *T* or the array access *ArrAcc* (*VarAcc a T′*) (*Lit* (*Prim* (*Intg* 2)) *T′′*) *T* are valid left-values, whereas term *Deref* (*Deref p T′*) *T* or the array access *ArrAcc* (*VarAcc a T′*) (*VarAcc i T′′*) *T* are still valid left-expressions but are not regarded as left-*values*. We define the predicate *reduced* to test whether a left-expression is a left-value. It tests whether only literal values appear as array indexes or as pointers in a dereferenced expression.

Definition 7.25 ►

$$
\begin{array}{lcl}
\textit{reduced} :: \textit{'a expr} \Rightarrow \textit{bool} & & \\
\textit{reduced}\ (\textit{VarAcc vname}\ T) & = & \textit{True} \\
\textit{reduced}\ (\textit{Deref}\ (\textit{Lit}\ a\ T')\ T) & = & \textit{True} \\
\textit{reduced}\ (\textit{ArrAcc lv}\ (\textit{Lit}\ i\ T')\ T) & = & \textit{reduced lv} \\
\textit{reduced}\ (\textit{StructAcc lv cn}\ T) & = & \textit{reduced lv} \\
\textit{reduced}\ \text{-} & = & \textit{False}
\end{array}
$$

Assignment of a value to a left-expression is performed in two steps. First the left value is calculated and then the new value is assigned into the state according to the left value.

Left-expression evaluation: leval L s e, of left-expression e in state s in context of local variables L is defined in Figure 7.5. ◀ Definition 7.26 *Left-expression evaluation*

$$\begin{array}{l}
\textit{leval} :: \textit{vname set} \Rightarrow \textit{state} \Rightarrow {}'a\ \textit{expr} \Rightarrow {}'a\ \textit{expr option}\\
\textit{leval}\ L\ s\ (\textit{VarAcc}\ vn\ T) = \lfloor \textit{VarAcc}\ vn\ T \rfloor\\
\textit{leval}\ L\ s\ (\textit{ArrAcc}\ e\ i\ T) =\\
\textsf{case}\ \textit{leval}\ L\ s\ e\ \textsf{of}\ \textit{None} \Rightarrow \textit{None}\\
\mid \lfloor ev \rfloor \Rightarrow \textsf{case}\ \textit{eval}\ L\ s\ i\ \textsf{of}\ \textit{None} \Rightarrow \textit{None} \mid \lfloor iv \rfloor \Rightarrow \lfloor \textit{ArrAcc}\ ev\ (\textit{Lit}\ iv\ (\textit{typ}\ i))\ T \rfloor\\
\textit{leval}\ L\ s\ (\textit{StructAcc}\ e\ cn\ T) =\\
\textsf{case}\ \textit{leval}\ L\ s\ e\ \textsf{of}\ \textit{None} \Rightarrow \textit{None} \mid \lfloor ev \rfloor \Rightarrow \lfloor \textit{StructAcc}\ ev\ cn\ T \rfloor\\
\textit{leval}\ L\ s\ (\textit{Deref}\ e\ T) =\\
\textsf{case}\ \textit{eval}\ L\ s\ e\ \textsf{of}\ \textit{None} \Rightarrow \textit{None} \mid \lfloor ev \rfloor \Rightarrow \lfloor \textit{Deref}\ (\textit{Lit}\ ev\ (\textit{typ}\ e))\ T \rfloor\\
\textit{leval}\ L\ s\ \text{-} = \textit{None}
\end{array}$$

Figure 7.5: Evaluation of C0 left-expressions

Left-expression evaluation *leval* produces *reduced* left-expressions.

If leval L s $le = \lfloor lv \rfloor$ then reduced lv. ◀ Lemma 7.1

Proof. By induction on *le*. □

Assignment: assign L s lv v, of value v to left-value lv in state s in context of local variables L is defined in Figure 7.6. ◀ Definition 7.27 *Assignment*

The assignment follows the path given via the left value *lv* and weaves the new value *v* into the current state. It also checks for array bound violations and dereferencing null pointers. The auxiliary function *null-lit* is defined as follows:

◀ Definition 7.28

$$\begin{array}{lcl}
\textit{null-lit} :: {}'a\ \textit{expr} \Rightarrow \textit{bool} & & \\
\textit{null-lit}\ (\textit{Lit}\ (\textit{Prim Null})\ T) & = & \textit{True}\\
\textit{null-lit}\ \text{-} & = & \textit{False}
\end{array}$$

The auxiliary function *assoc-update* preforms an update in an assocation list.

◀ Definition 7.29

$$\begin{array}{lcl}
\textit{assoc-update} :: ({}'k \times {}'v)\ \textit{list} \Rightarrow {}'k \Rightarrow {}'v \Rightarrow ({}'k \times {}'v)\ \textit{list} & & \\
\textit{assoc-update}\ []\ k\ v & = & []\\
\textit{assoc-update}\ (x{\cdot}xs)\ k\ v & = & \textsf{if}\ k = \textit{fst}\ x\ \textsf{then}\ (k, v){\cdot}xs\ \textsf{else}\ x{\cdot}\textit{assoc-update}\ xs\ k\ v
\end{array}$$

We also provide a lifted version of assignment that watches for faults in the left-value and value argument:

◀ Definition 7.30

$$\begin{array}{lcl}
\textit{assign-opt} :: \textit{vname set} \Rightarrow \textit{state} \Rightarrow {}'a\ \textit{expr option} \Rightarrow \textit{val option} \Rightarrow \textit{state option} & & \\
\textit{assign-opt}\ L\ s\ \textit{None}\ vo & = & \textit{None}\\
\textit{assign-opt}\ L\ s\ \lfloor e \rfloor\ vo & = & \textsf{case}\ vo\ \textsf{of}\ \textit{None} \Rightarrow \textit{None} \mid \lfloor v \rfloor \Rightarrow \textit{assign}\ L\ s\ e\ v
\end{array}$$

```
assign :: vname set ⇒ state ⇒ 'a expr ⇒ val ⇒ state option

assign L s (VarAcc vn T) v = ⌊update-var L s vn v⌋

assign L s (StructAcc lv fn T) v =
case eval L s lv of None ⇒ None
| ⌊ev⌋ ⇒
  let fs = the-Structv ev;
      new-val = Structv (assoc-update fs fn v)
  in if fn ∈ set (map fst fs) then assign L s lv new-val else None

assign L s (ArrAcc lv i T) v =
case eval L s lv of None ⇒ None
| ⌊ev⌋ ⇒
  let a = the-Arrv ev;
      n = the-Unsgnd_v (the-Lit i)
  in if n < |a| then assign L s lv (Arrv (a[n := v])) else None

assign L s (Deref lt T) v =
if null-lit lt then None
else ⌊s(|heap := heap s(the-Addr_v (the-Lit lt) ↦ v)|)⌋
assign L s - - = None
```

Figure 7.6: Assignment of a value to a left-value

7.2.4 Big-Step Semantics

Definition 7.31 ► *Big-step semantics of C0*

The operational big-step semantics: $\Pi,L\vdash_{C0} \langle c,s\rangle \Rightarrow t$, is defined inductively by the rules in Figure 7.7. In program Π and in context of local variables L execution of command c transforms the initial state s to the final state t, where:

Π :: *'a prog*	*s,t* :: *state option*
L :: *vname set*	*c* :: *'a stmt*

Like the big-step semantics of Simpl (cf. Definition 2.4) the rules are divided into two parts. The syntax directed ones are only applicable to normal states $\lfloor s\rfloor$, whereas the fault propagation rule is applicable for all commands and skips execution if the state is *None*.

The *Skip* statement leaves the state unmodified.

An assignment *Ass le e* evaluates the left-expression *le* and the expression *e* and assigns its value.

For a pointer allocation *PAlloc le tn* first the type corresponding to the type-name *tn* is obtained. Then a new heap location is allocated. If heap allocation fails the *Null* pointer is assigned to the left-value obtained from left-expression *le*. Otherwise we obtain a fresh location *l*, initialise it with the proper default value and decrement the *free-heap* counter. Finally we assign the location *l* to the left-value obtained from left-expression *le*. Heap allocation is performed by the auxiliary function *new-Addr*. It considers *free-heap* to decide whether there is enough memory to allocate the requested object.

If *new-Addr f T h* = $\lfloor l\rfloor$ then *h l* = *None*, whenever the domain of heap *h* is finite. This means that location *l* is "fresh" with respect to heap *h*.

$$\frac{}{\Pi,L\vdash_{C0} \langle \mathit{Skip},\lfloor s\rfloor\rangle \Rightarrow \lfloor s\rfloor}\;(\textsc{Skip})\qquad \frac{t = \mathit{assign\text{-}opt}\ L\ s\ (\mathit{leval}\ L\ s\ le)\ (\mathit{eval}\ L\ s\ e)}{\Pi,L\vdash_{C0} \langle \mathit{Ass}\ le\ e,\lfloor s\rfloor\rangle \Rightarrow t}\;(\textsc{Assignment})$$

$$\frac{\begin{array}{c}\mathit{tnenv}\ \Pi\ tn = \lfloor T\rfloor \qquad \mathit{new\text{-}Addr}\ (\mathit{free\text{-}heap}\ s)\ T\ (\mathit{heap}\ s) = \lfloor l\rfloor \\ s_1 = s(\!|\mathit{heap} := \mathit{heap}\ s(l \mapsto \mathit{default\text{-}val}\ T), \mathit{free\text{-}heap} := \mathit{free\text{-}heap}\ s - \mathit{sizeof\text{-}type}\ T|\!) \\ t = \mathit{assign\text{-}opt}\ L\ s_1\ (\mathit{leval}\ L\ s\ le)\ \lfloor \mathit{Prim}\ (\mathit{Addr}\ l)\rfloor\end{array}}{\Pi,L\vdash_{C0} \langle \mathit{PAlloc}\ le\ tn,\lfloor s\rfloor\rangle \Rightarrow t}\;(\textsc{PAlloc})$$

$$\frac{\begin{array}{c}\mathit{tnenv}\ \Pi\ tn = \lfloor T\rfloor \\ \mathit{new\text{-}Addr}\ (\mathit{free\text{-}heap}\ s)\ T\ (\mathit{heap}\ s) = \mathit{None} \qquad t = \mathit{assign\text{-}opt}\ L\ s\ (\mathit{leval}\ L\ s\ le)\ \lfloor \mathit{Prim}\ \mathit{Null}\rfloor\end{array}}{\Pi,L\vdash_{C0} \langle \mathit{PAlloc}\ le\ tn,\lfloor s\rfloor\rangle \Rightarrow t}\;(\textsc{PAllocFail})$$

$$\frac{\Pi,L\vdash_{C0} \langle c_1,\lfloor s\rfloor\rangle \Rightarrow s_1 \qquad \Pi,L\vdash_{C0} \langle c_2,s_1\rangle \Rightarrow s_2}{\Pi,L\vdash_{C0} \langle \mathit{Comp}\ c_1\ c_2,\lfloor s\rfloor\rangle \Rightarrow s_2}\;(\textsc{Comp})$$

$$\frac{\begin{array}{c}\mathit{eval}\ L\ s\ e = \lfloor \mathit{Prim}\ (\mathit{Bool}\ \mathit{True})\rfloor \\ \Pi,L\vdash_{C0} \langle c_1,\lfloor s\rfloor\rangle \Rightarrow t\end{array}}{\Pi,L\vdash_{C0} \langle \mathit{Ifte}\ e\ c_1\ c_2,\lfloor s\rfloor\rangle \Rightarrow t}\;(\textsc{IfteTrue})\qquad \frac{\begin{array}{c}\mathit{eval}\ L\ s\ e = \lfloor \mathit{Prim}\ (\mathit{Bool}\ \mathit{False})\rfloor \\ \Pi,L\vdash_{C0} \langle c_2,\lfloor s\rfloor\rangle \Rightarrow t\end{array}}{\Pi,L\vdash_{C0} \langle \mathit{Ifte}\ e\ c_1\ c_2,\lfloor s\rfloor\rangle \Rightarrow t}\;(\textsc{IfteFalse})$$

$$\frac{\mathit{eval}\ L\ s\ e = \mathit{None}}{\Pi,L\vdash_{C0} \langle \mathit{Ifte}\ e\ c_1\ c_2,\lfloor s\rfloor\rangle \Rightarrow \mathit{None}}\;(\textsc{IfteFail})$$

$$\frac{\mathit{eval}\ L\ s\ e = \lfloor \mathit{Prim}\ (\mathit{Bool}\ \mathit{True})\rfloor \qquad \Pi,L\vdash_{C0} \langle c,\lfloor s\rfloor\rangle \Rightarrow s_1 \qquad \Pi,L\vdash_{C0} \langle \mathit{Loop}\ e\ c,s_1\rangle \Rightarrow t}{\Pi,L\vdash_{C0} \langle \mathit{Loop}\ e\ c,\lfloor s\rfloor\rangle \Rightarrow t}\;(\textsc{LoopTrue})$$

$$\frac{\mathit{eval}\ L\ s\ e = \lfloor \mathit{Prim}\ (\mathit{Bool}\ \mathit{False})\rfloor}{\Pi,L\vdash_{C0} \langle \mathit{Loop}\ e\ c,\lfloor s\rfloor\rangle \Rightarrow \lfloor s\rfloor}\;(\textsc{LoopFalse})\qquad \frac{\mathit{eval}\ L\ s\ e = \mathit{None}}{\Pi,L\vdash_{C0} \langle \mathit{Loop}\ e\ c,\lfloor s\rfloor\rangle \Rightarrow \mathit{None}}\;(\textsc{LoopFail})$$

$$\frac{\begin{array}{c}\mathit{plookup}\ \Pi\ pn = \lfloor((pds,\ lds,\ rT),\ body)\rfloor \\ pns = \mathit{map}\ \mathit{fst}\ pds \qquad lns = \mathit{map}\ \mathit{fst}\ lds \qquad L' = \mathit{set}\ (pns\ @\ lns\ @\ [\mathit{Res}]) \\ \Pi,L'\vdash_{C0} \langle body,\mathit{set\text{-}locv\text{-}new\text{-}frame}\ s\ pns\ (\mathit{evals}\ L\ s\ ps)\rangle \Rightarrow t\end{array}}{\Pi,L\vdash_{C0} \langle \mathit{SCall}\ vn\ pn\ ps,\lfloor s\rfloor\rangle \Rightarrow \mathit{reset\text{-}locv\text{-}old\text{-}frame}\ L\ s\ t\ vn}\;(\textsc{SCall})$$

$$\frac{t = \mathit{assign\text{-}opt}\ L\ s\ \lfloor \mathit{VarAcc}\ \mathit{Res}\ (\mathit{typ}\ e)\rfloor\ (\mathit{eval}\ L\ s\ e)}{\Pi,L\vdash_{C0} \langle \mathit{Return}\ e,\lfloor s\rfloor\rangle \Rightarrow t}\;(\textsc{Return})$$

$$\frac{}{\Pi,L\vdash_{C0} \langle c,\mathit{None}\rangle \Rightarrow \mathit{None}}\;(\textsc{FaultProp})$$

Figure 7.7: Big-step execution rules for C0

The definition of *new-Addr* builds on *new* for references (cf. Definition 2.12). The functions *Rep-loc* and *Abs-loc* convert between types *ref* and *loc*.

◂ Definition 7.32

$\mathit{new\text{-}Addr} :: \mathit{nat} \Rightarrow \mathit{ty} \Rightarrow (\mathit{loc} \rightharpoonup \mathit{val}) \Rightarrow \mathit{loc}\ \mathit{option}$
$\mathit{new\text{-}Addr}\ \mathit{free}\ T\ h \equiv$
if $\mathit{sizeof\text{-}type}\ T \le \mathit{free}$ then $\lfloor \mathit{Abs\text{-}loc}\ (\mathit{new}\ (\mathit{Rep\text{-}loc}\ `\ \mathit{dom}\ h))\rfloor$ else None

The rules for sequential composition *Comp* c_1 c_2, conditional execution *Ifte* e c_1 c_2 and the loop *Loop* e c are standard. If a runtime fault occurs during evaluation of the branch condition e it is propagated.

To execute a procedure call *SCall vn pn ps* first the procedure definition of procedure *pn* is retrieved: ((*pds*, *lds*, *rT*), *body*). The names of the formal parameters *pns* and local variables *lns* are extracted from this definition. Additionally to these names there is the reserved name *Res* for the result variable. The procedure body assigns the result value to this variable. Altogether the variables *pns*, *lns* and *Res* are regarded as local variables for the procedure body. Before entering the procedure body the actual parameters *ps* are evaluated and are used to initialise the new frame for the procedure body:

Definition 7.33 ►

$$\begin{array}{l} \textit{set-locv-new-frame} :: \textit{state} \Rightarrow \textit{vname list} \Rightarrow \textit{val list option} \Rightarrow \textit{state option} \\ \textit{set-locv-new-frame } s \textit{ pns None} = \textit{None} \\ \textit{set-locv-new-frame } s \textit{ pns } \lfloor pvs \rfloor = \lfloor s(\!|lvars := [pns\ [\mapsto]\ pvs]|\!) \rfloor \end{array}$$

Only the parameters are initialised. The local variables and the result variables remain uninitialised. After the body is executed the local variables of the caller are restored and the content of the result variable *Res* is copied to variable *vn*.

Definition 7.34 ►

$$\begin{array}{ll} \multicolumn{2}{l}{\textit{reset-locv-old-frame} :: \textit{vname set} \Rightarrow \textit{state} \Rightarrow \textit{state option} \Rightarrow \textit{vname} \Rightarrow \textit{state option}} \\ \multicolumn{2}{l}{\textit{reset-locv-old-frame } L\ s \textit{ None } vn = \textit{None}} \\ \textit{reset-locv-old-frame } L\ s\ \lfloor t \rfloor\ vn & = \textsf{case } \textit{lvars } t \textit{ Res } \textsf{of } \textit{None} \Rightarrow \textit{None} \\ & \quad\ \mid \lfloor r \rfloor \Rightarrow \lfloor \textit{update-var } L\ (t(\!|lvars := lvars\ s|\!))\ vn\ r \rfloor \end{array}$$

The return statement *Return e* is a mere abbreviation for the assignment to the result variable *Res*. It does not exit the procedure immediately.

7.2.5 Termination

Analogous to the Simpl (cf. Definition 2.5) we define a termination judgement for C0 programs.

Definition 7.35 ► *Guaranteed termination of C0*

Guaranteed termination: $\Pi,L\vdash_{C0}\ c \downarrow s$, *of statement* c *in the initial state* s *within the context of program* Π *and local variables* L *is defined inductively by the rules in Figure 7.8, where:*

$$\begin{array}{ll} \Pi :: \textit{'a prog} & s :: \textit{state option} \\ L :: \textit{vname set} & c :: \textit{'a stmt} \end{array}$$

If statement c terminates when started in state s, then there is a final state t according to the big-step semantics.

Lemma 7.2 ► *If* $\Pi,L\vdash_{C0}\ c \downarrow s$ *then* $\exists t.\ \Pi,L\vdash_{C0}\ \langle c,s\rangle \Rightarrow t$.

Proof. By induction on the termination judgement. □

In contrast to Simpl, C0 is deterministic:

Lemma 7.3 ► *If* $\Pi,L\vdash_{C0}\ \langle c,s\rangle \Rightarrow t$ *and* $\Pi,L\vdash_{C0}\ \langle c,s\rangle \Rightarrow t'$ *then* $t = t'$.

Proof. By induction on the execution $\Pi,L\vdash_{C0}\ \langle c,s\rangle \Rightarrow t$. □

Hence we also get termination from a big-step execution:

Lemma 7.4 ► *If* $\Pi,L\vdash_{C0}\ \langle c,s\rangle \Rightarrow t$ *then* $\Pi,L\vdash_{C0}\ c \downarrow s$.

$$\dfrac{}{\Pi,L\vdash_{C0}\ \mathit{Skip}\downarrow\lfloor s\rfloor}\ (\textsc{Skip})\qquad \dfrac{}{\Pi,L\vdash_{C0}\ \mathit{Ass\ le\ e}\downarrow\lfloor s\rfloor}\ (\textsc{Assignment})\qquad \dfrac{\mathit{tnenv}\ \Pi\ \mathit{tn}=\lfloor T\rfloor}{\Pi,L\vdash_{C0}\ \mathit{PAlloc\ le\ tn}\downarrow\lfloor s\rfloor}\ (\textsc{PAlloc})$$

$$\dfrac{\Pi,L\vdash_{C0}\ c_1\downarrow\lfloor s\rfloor\qquad \forall s'.\ \Pi,L\vdash_{C0}\ \langle c_1,\lfloor s\rfloor\rangle\Rightarrow s'\longrightarrow \Pi,L\vdash_{C0}\ c_2\downarrow s'}{\Pi,L\vdash_{C0}\ \mathit{Comp}\ c_1\ c_2\downarrow\lfloor s\rfloor}\ (\textsc{Comp})$$

$$\dfrac{\mathit{eval}\ L\ s\ e=\lfloor \mathit{Prim}\ (\mathit{Bool\ True})\rfloor\qquad \Pi,L\vdash_{C0}\ c_1\downarrow\lfloor s\rfloor}{\Pi,L\vdash_{C0}\ \mathit{Ifte}\ e\ c_1\ c_2\downarrow\lfloor s\rfloor}\ (\textsc{IfteTrue})$$

$$\dfrac{\mathit{eval}\ L\ s\ e=\lfloor \mathit{Prim}\ (\mathit{Bool\ False})\rfloor\qquad \Pi,L\vdash_{C0}\ c_2\downarrow\lfloor s\rfloor}{\Pi,L\vdash_{C0}\ \mathit{Ifte}\ e\ c_1\ c_2\downarrow\lfloor s\rfloor}\ (\textsc{IfteFalse})\qquad \dfrac{\mathit{eval}\ L\ s\ e=\mathit{None}}{\Pi,L\vdash_{C0}\ \mathit{Ifte}\ e\ c_1\ c_2\downarrow\lfloor s\rfloor}\ (\textsc{IfteFail})$$

$$\dfrac{\begin{array}{c}\mathit{eval}\ L\ s\ e=\lfloor \mathit{Prim}\ (\mathit{Bool\ True})\rfloor\\ \Pi,L\vdash_{C0}\ c\downarrow\lfloor s\rfloor\qquad \forall s'.\ \Pi,L\vdash_{C0}\ \langle c,\lfloor s\rfloor\rangle\Rightarrow s'\longrightarrow \Pi,L\vdash_{C0}\ \mathit{Loop}\ e\ c\downarrow s'\end{array}}{\Pi,L\vdash_{C0}\ \mathit{Loop}\ e\ c\downarrow\lfloor s\rfloor}\ (\textsc{LoopTrue})$$

$$\dfrac{\mathit{eval}\ L\ s\ e=\lfloor \mathit{Prim}\ (\mathit{Bool\ False})\rfloor}{\Pi,L\vdash_{C0}\ \mathit{Loop}\ e\ c\downarrow\lfloor s\rfloor}\ (\textsc{LoopFalse})\qquad \dfrac{\mathit{eval}\ L\ s\ e=\mathit{None}}{\Pi,L\vdash_{C0}\ \mathit{Loop}\ e\ c\downarrow\lfloor s\rfloor}\ (\textsc{LoopFail})$$

$$\dfrac{\begin{array}{c}\mathit{plookup}\ \Pi\ \mathit{pn}=\lfloor \mathit{pdf}\rfloor\qquad \mathit{pns}=\mathit{map\ fst}\ (\mathit{pardecls\text{-}of}\ (\mathit{pdecl\text{-}of\ pdf}))\\ \mathit{lns}=\mathit{map\ fst}\ (\mathit{locdecls\text{-}of}\ (\mathit{pdecl\text{-}of\ pdf}))\qquad L'=\mathit{set}\ (\mathit{pns}\ @\ \mathit{lns}\ @\ [\mathit{Res}])\\ \Pi,L'\vdash_{C0}\ \mathit{pbody\text{-}of\ pdf}\downarrow \mathit{set\text{-}locv\text{-}new\text{-}frame}\ s\ \mathit{pns}\ (\mathit{evals}\ L\ s\ \mathit{ps})\end{array}}{\Pi,L\vdash_{C0}\ \mathit{SCall}\ \mathit{vn}\ \mathit{pn}\ \mathit{ps}\downarrow\lfloor s\rfloor}\ (\textsc{SCall})$$

$$\dfrac{}{\Pi,L\vdash_{C0}\ \mathit{Return}\ e\downarrow\lfloor s\rfloor}\ (\textsc{Return})$$

$$\dfrac{}{\Pi,L\vdash_{C0}\ c\downarrow \mathit{None}}\ (\textsc{Fault})$$

Figure 7.8: Guaranteed termination for C0

Proof. By induction on the execution $\Pi,L\vdash_{C0}\ \langle c,s\rangle\Rightarrow t$ and Lemma 7.3 to handle the intermediate states for sequential composition and the loop. □

Altogether we have the following equivalence between termination and the big-step semantics:

$$\Pi,L\vdash_{C0}\ c\downarrow s=(\exists t.\ \Pi,L\vdash_{C0}\ \langle c,s\rangle\Rightarrow t)$$

◂ Lemma 7.5
Terminates iff exists execution

Proof. By Lemmas 7.2 and 7.4. □

7.2.6 Typing

C0 is a statically typed language. As we have seen in the definition of the operational semantics in the previous section there are no dynamic type checks. Statically welltyped programs also behave type correct as they are executed. This is formally proven in the next section. This property is crucial for correctness of the translation to Simpl. In Simpl the state is already implicitly typed by the translation to a state-record. The correspondence to C0 programs is only guaranteed for welltyped programs. To avoid redundancy, the type-system we introduce is generalised to work both for static typing and to describe the type invariant that holds upon execution of a C0 program.

Wellformed types We start with wellformedness of types. All pointer types have to be declared and the field names of structures have to be unique.

Definition 7.36 ► $\mathit{unique} :: ('a \times 'b)\ \mathit{list} \Rightarrow \mathit{bool}\ \mathit{unique} \equiv \mathit{distinct} \circ \mathit{map}\ \mathit{fst}$

Definition 7.37 ► *Wellformed types* *The judgement* $TE \vdash T \surd$ *expresses that type* $T :: ty$ *is wellformed with respect to a type environment* $TE :: tname \rightharpoonup ty$*. It is defined inductively by the rules in Figure 7.9*

$$\frac{}{TE \vdash \mathit{Boolean} \surd} \qquad \frac{}{TE \vdash \mathit{Integer} \surd} \qquad \frac{}{TE \vdash \mathit{UnsgndT} \surd} \qquad \frac{}{TE \vdash \mathit{CharT} \surd}$$

$$\frac{}{TE \vdash \mathit{NullT} \surd} \qquad \frac{TE\ tn = \lfloor T \rfloor}{TE \vdash \mathit{Ptr}\ tn \surd}$$

$$\frac{TE \vdash T \surd}{TE \vdash \mathit{Arr}\ n\ T \surd} \qquad \frac{\mathit{unique}\ fTs \qquad \forall T \in \mathit{set}\ (\mathit{map}\ \mathit{snd}\ fTs).\ TE \vdash T \surd}{TE \vdash \mathit{Struct}\ fTs \surd}$$

Figure 7.9: Wellformed types

Typing of values As introduced in Section 7.1.2 pointers in C0 are typed. If we are interested in the type of an address, a *heap typing* that maps locations to types can be provided to the typing judgement. If no heap typing is given every address *Addr l* fits to any pointer type *Ptr tn*.

Definition 7.38 ► *Typing of values* *The judgement* $HT \vdash_v v :: T$ *expresses that value* v *is compatible with type* $T :: ty$ *with respect to an optional heap typing* $HT :: (loc \rightharpoonup tname)\ option$*. It is defined inductively by the rules in Figure 7.10. If* $HT = None$ *it can be omitted.*

$$\frac{}{HT \vdash_v \mathit{Prim}\ (\mathit{Bool}\ b) :: \mathit{Boolean}} \qquad \frac{\mathit{int\text{-}lb} \le i \qquad i < \mathit{int\text{-}ub}}{HT \vdash_v \mathit{Prim}\ (\mathit{Intg}\ i) :: \mathit{Integer}} \qquad \frac{n < \mathit{un\text{-}int\text{-}ub}}{HT \vdash_v \mathit{Prim}\ (\mathit{Unsgnd}\ n) :: \mathit{UnsgndT}}$$

$$\frac{\mathit{chr\text{-}lb} \le i \qquad i < \mathit{chr\text{-}ub}}{HT \vdash_v \mathit{Prim}\ (\mathit{Chr}\ i) :: \mathit{CharT}} \qquad \frac{}{HT \vdash_v \mathit{Prim}\ \mathit{Null} :: \mathit{Ptr}\ tn} \qquad \frac{}{HT \vdash_v \mathit{Prim}\ \mathit{Null} :: \mathit{NullT}}$$

$$\frac{HT\ l = \lfloor tn \rfloor}{\lfloor HT \rfloor \vdash_v \mathit{Prim}\ (\mathit{Addr}\ l) :: \mathit{Ptr}\ tn} \qquad \frac{}{\vdash_v \mathit{Prim}\ (\mathit{Addr}\ l) :: \mathit{Ptr}\ tn}$$

$$\frac{n = |vs| \qquad \forall v \in \mathit{set}\ vs.\ HT \vdash_v v :: T}{HT \vdash_v \mathit{Arrv}\ vs :: \mathit{Arr}\ n\ T} \qquad \frac{\begin{array}{c}\mathit{map}\ \mathit{fst}\ \mathit{fvs} = \mathit{map}\ \mathit{fst}\ fTs \\ \forall (v, T) \in \mathit{set}\ (\mathit{zip}\ (\mathit{map}\ \mathit{snd}\ \mathit{fvs})\ (\mathit{map}\ \mathit{snd}\ fTs)).\ HT \vdash_v v :: T\end{array}}{HT \vdash_v \mathit{Structv}\ \mathit{fvs} :: \mathit{Struct}\ fTs}$$

Figure 7.10: Typing of values

Values of a numeric type have to respect their bounds.

The *Null* pointer fits to any pointer type *Ptr tn* and to the type *NullT*.

If a heap typing $\lfloor HT \rfloor$ is provided and $HT\ l = \lfloor tn \rfloor$, then an address *Addr l* only fits to the pointer type *Ptr tn*. If no heap typing is supplied it fits to any pointer type. For static typing of C0 expressions, literal values may not contain addresses. This can be enforced by providing the empty heap typing $empty \equiv \lambda l.\ None$ to the typing judgement. $\lfloor empty \rfloor \vdash_v Prim\ (Addr\ l) :: Ptr\ tn$ is never valid and thus prevents literal addresses to appear in C0 source code.

For array values, the array size has to fit to the type and the values stored in the array have to be typed correctly.

For structures the field-names have to coincide with the field-names of the type and the values have to be typed correctly.

Typing of expressions Typing of expressions is defined for type-tagged expressions *ty expr*. Hence no explicit type occurs in the typing judgement, which is of the form $\Pi,VT,HT \vdash_e e\ \surd$. The heap typing *HT* is used to type literal values, *VT* is a type environment for variable names and Π is the program. Actually the only relevant information from the program is the type-name environment *tnenv* Π.

◂ Definition 7.39 *Typing of expressions*

Typing of expressions: $\Pi,VT,HT \vdash_e e\ \surd$, *with respect to program* Π, *type environment VT for variables and heap typing HT is defined inductively by the rules in Figure 7.11. Expression lists are handled by judgement* $\Pi,VT,HT [\vdash_e]\ es\ \surd$. *Left-expressions are typed according to* $\Pi,VT,HT \vdash_l e\ \surd$ *(cf. Figure 7.12). Where:*

$$\begin{array}{ll} \Pi :: ty\ prog & e :: ty\ expr \\ VT :: vname \rightharpoonup ty & es :: ty\ expr\ list \\ HT :: loc \rightharpoonup tname & \end{array}$$

Literal values *Lit v T* are welltyped if *v* is a welltyped value and type *T* is wellformed.

For a variable access *VarAcc vn T* the type has to conform to the type environment *VT* and the type has to be wellformed.

An array access *ArrAcc a i T* is welltyped provided that *a* is an array and *i* an unsigned integer.

If *e* is a structure and field *fn* has type *T*, then the structure access *StructAcc e fn T* is welltyped.

Dereferencing a pointer *Deref e T* is accepted, if *e* is a pointer to type-name *tn* that is declared with the wellformed type *T* in the program.

Unary and binary operations are welltyped, if they conform to the auxiliary rules $\ll uop \gg T_1 :: T$ and $T_1 \ll bop \gg T_2 :: T$ respectively. The offset of shift operations is restricted to unsigned integers. The order relations are restricted to numeric types. Structures, arrays and pointers are excluded. Equality is only defined for primitive types. These include pointers but not structures or arrays. The lazy binary operations are only defined for Booleans.

Left expressions share the same typing rules as expressions but are restricted to variable-, array- and structure-access and dereferencing pointers.

As mentioned for the typing of values, static typing is achieved by setting the heap typing *HT* to the empty environment. This prohibits literal address values in C0 sources. By induction on the typing judgement, we get that a welltyped expression has a wellformed type.

◂ Lemma 7.6

If $\Pi,VT,HT \vdash_e e\ \surd$ *then* $tnenv\ \Pi \vdash typ\ e\ \surd$.

$$\frac{\lfloor HT \rfloor \vdash_v v :: T \qquad tnenv\ \Pi \vdash T\ \surd}{\Pi, VT, HT \vdash_e Lit\ v\ T\ \surd} \qquad \frac{VT\ vn = \lfloor T \rfloor \qquad tnenv\ \Pi \vdash T\ \surd}{\Pi, VT, HT \vdash_e VarAcc\ vn\ T\ \surd}$$

$$\frac{\begin{array}{c}\Pi, VT, HT \vdash_e a\ \surd \qquad typ\ a = Arr\ n\ T \\ \Pi, VT, HT \vdash_e i\ \surd \qquad typ\ i = UnsgndT\end{array}}{\Pi, VT, HT \vdash_e ArrAcc\ a\ i\ T\ \surd} \qquad \frac{\begin{array}{c}\Pi, VT, HT \vdash_e e\ \surd \\ typ\ e = Struct\ fs \qquad map\text{-}of\ fs\ fn = \lfloor T \rfloor\end{array}}{\Pi, VT, HT \vdash_e StructAcc\ e\ fn\ T\ \surd}$$

$$\frac{\Pi, VT, HT \vdash_e e\ \surd \qquad typ\ e = Ptr\ tn \qquad tnenv\ \Pi\ tn = \lfloor T \rfloor \qquad tnenv\ \Pi \vdash T\ \surd}{\Pi, VT, HT \vdash_e Deref\ e\ T\ \surd}$$

$$\frac{\Pi, VT, HT \vdash_e e\ \surd \qquad \ll uop \gg typ\ e :: T}{\Pi, VT, HT \vdash_e UnOp\ uop\ e\ T\ \surd} \qquad \frac{\Pi, VT, HT \vdash_e e_1\ \surd \qquad \Pi, VT, HT \vdash_e e_2\ \surd \qquad typ\ e_1 \ll bop \gg typ\ e_2 :: T}{\Pi, VT, HT \vdash_e BinOp\ bop\ e_1\ e_2\ T\ \surd}$$

$$\frac{\Pi, VT, HT \vdash_e e_1\ \surd \qquad \Pi, VT, HT \vdash_e e_2\ \surd \qquad typ\ e_1 = Boolean \qquad typ\ e_2 = Boolean}{\Pi, VT, HT \vdash_e LazyBinOp\ bop\ e_1\ e_2\ Boolean\ \surd}$$

$$\frac{}{\Pi, VT, HT [\vdash_e]\ []\ \surd} \qquad \frac{\Pi, VT, HT \vdash_e e\ \surd \qquad \Pi, VT, HT [\vdash_e]\ es\ \surd}{\Pi, VT, HT [\vdash_e]\ e \cdot es\ \surd}$$

$$\frac{T \in \{Integer,\ CharT\}}{\ll unary\text{-}minus \gg T :: T} \qquad \frac{numeric\text{-}type\ T}{\ll bitwise\text{-}neg \gg T :: T} \qquad \frac{}{\ll logical\text{-}not \gg Boolean :: Boolean}$$

$$\frac{numeric\text{-}type\ T}{\ll to\text{-}int \gg T :: Integer} \qquad \frac{numeric\text{-}type\ T}{\ll to\text{-}unsigned\text{-}int \gg T :: UnsgndT} \qquad \frac{numeric\text{-}type\ T}{\ll to\text{-}char \gg T :: CharT}$$

$$\frac{numeric\text{-}type\ T \qquad bop \in \{plus,\ minus,\ times,\ divides,\ bitwise\text{-}or,\ bitwise\text{-}and,\ bitwise\text{-}xor\}}{T \ll bop \gg T :: T}$$

$$\frac{\begin{array}{c}numeric\text{-}type\ T \\ bop \in \{shiftleft,\ shiftright\}\end{array}}{T \ll bop \gg UnsgndT :: T} \qquad \frac{\begin{array}{c}numeric\text{-}type\ T \\ bop \in \{greater,\ less,\ greaterequal,\ lessequal\}\end{array}}{T \ll bop \gg T :: Boolean} \qquad \frac{\begin{array}{c}prim\text{-}type\ T_1 \qquad prim\text{-}type\ T_2 \\ T_1 \leq T_2 \vee T_2 \leq T_1 \\ bop \in \{equal,\ notequal\}\end{array}}{T_1 \ll bop \gg T_2 :: Boolean}$$

Figure 7.11: Typing of expressions

$$\frac{\Pi, VT, HT \vdash_e VarAcc\ vn\ T\ \surd}{\Pi, VT, HT \vdash_l VarAcc\ vn\ T\ \surd} \qquad \frac{\Pi, VT, HT \vdash_l e\ \surd \qquad typ\ e = Struct\ fs \qquad map\text{-}of\ fs\ fn = \lfloor T \rfloor}{\Pi, VT, HT \vdash_l StructAcc\ e\ fn\ T\ \surd}$$

$$\frac{\Pi, VT, HT \vdash_l a\ \surd \qquad typ\ a = Arr\ n\ T \qquad \Pi, VT, HT \vdash_e i\ \surd \qquad typ\ i = UnsgndT}{\Pi, VT, HT \vdash_l ArrAcc\ a\ i\ T\ \surd} \qquad \frac{\Pi, VT, HT \vdash_e Deref\ e\ T\ \surd}{\Pi, VT, HT \vdash_l Deref\ e\ T\ \surd}$$

Figure 7.12: Typing of left-expressions

Typing of Statements

◂ Definition 7.40
Typing of statements

The judgement $\Pi,VT,HT \vdash c \surd$ *ensures that statement* c *is welltyped with respect to program* Π*, type environment* VT *for variables and heap typing* HT*. It is defined inductively by the rules in Figure 7.13, where*

$$\begin{array}{ll} \Pi :: ty\ prog & HT :: loc \rightharpoonup tname \\ VT :: vname \rightharpoonup ty & c :: ty\ stmt \end{array}$$

$$\frac{}{\Pi,VT,HT \vdash Skip \surd} \qquad \frac{\Pi,VT,HT \vdash_l le \surd \qquad \Pi,VT,HT \vdash_e e \surd \qquad typ\ e \le typ\ le}{\Pi,VT,HT \vdash Ass\ le\ e \surd}$$

$$\frac{\Pi,VT,HT \vdash_l le \surd \qquad typ\ le = Ptr\ tn \qquad tnenv\ \Pi\ tn = \lfloor T \rfloor}{\Pi,VT,HT \vdash PAlloc\ le\ tn \surd} \qquad \frac{\Pi,VT,HT \vdash c_1 \surd \qquad \Pi,VT,HT \vdash c_2 \surd}{\Pi,VT,HT \vdash Comp\ c_1\ c_2 \surd}$$

$$\frac{\Pi,VT,HT \vdash_e e \surd \qquad typ\ e = Boolean \qquad \Pi,VT,HT \vdash c_1 \surd \qquad \Pi,VT,HT \vdash c_2 \surd}{\Pi,VT,HT \vdash Ifte\ e\ c_1\ c_2 \surd}$$

$$\frac{\Pi,VT,HT \vdash_e e \surd \qquad typ\ e = Boolean \qquad \Pi,VT,HT \vdash c \surd}{\Pi,VT,HT \vdash Loop\ e\ c \surd}$$

$$\frac{\begin{array}{c} plookup\ \Pi\ pn = \lfloor((pds, lds, rT), body)\rfloor \\ \Pi,VT,HT [\vdash_e] ps \surd \qquad map\ typ\ ps\ [\le]\ map\ snd\ pds \qquad VT\ vn = \lfloor rT \rfloor \end{array}}{\Pi,VT,HT \vdash SCall\ vn\ pn\ ps \surd}$$

$$\frac{VT\ Res = \lfloor T \rfloor \qquad tnenv\ \Pi \vdash T \surd \qquad \Pi,VT,HT \vdash_e e \surd \qquad typ\ e \le T}{\Pi,VT,HT \vdash Return\ e \surd}$$

....

$$\frac{}{T \le T} \qquad \frac{}{NullT \le Ptr\ tn}$$

$$\frac{T \le Q}{Arr\ n\ T \le Arr\ n\ Q} \qquad \frac{map\ fst\ fTs = map\ fst\ fQs \qquad \forall (T, Q) \in set\ (zip\ (map\ snd\ fTs)\ (map\ snd\ fQs)).\ T \le Q}{Struct\ fTs \le Struct\ fQs}$$

$$Ts\ [\le]\ Qs \equiv |Ts| = |Qs| \wedge (\forall (T, Q) \in set\ (zip\ Ts\ Qs).\ T \le Q)$$

Figure 7.13: Typing of statements

Skip is welltyped.

To type an assignment *Ass le e* we introduce a widening relation on types. The type of expression *e* has to widen to the type of the left-expression *le*. C0 does not allow widening between numeric types like characters and integers. Explicit type casts have to be inserted there. The widening relation only supports a liberal typing of the *Null* pointer. It can always be typed with *NullT* instead of a concrete pointer type *Ptr tn*, for example, in an assignment like `v = null`. Technically we do not need this widening relation. We could omit the null type *NullT* completely, but this complicates the type elaboration phase. It then has to infer a proper pointer type for every null pointer. The widening relation is reflexive and *NullT* ≤ *Ptr tn*. For structure and array types the widening relation is inherited from the components.

Note that the covariant widening of array types is no issue for this very restricted widening. In the end it only permits that a null pointer can be read from or written to an array cell. The relation *Ts* [≤] *Qs* extends the widening relation to lists.

For pointer allocation *PAlloc le tn* the type of *le* has to coincide with *tn* and *tn* has to be defined in the type environment.

Typing of sequential composition, conditional execution and the loop is as expected.

For procedure calls *SCall vn pn ps* the types of the actual parameters have to widen to the types of the formal ones and the return type has to coincide with the type of variable *vn*.

For *Return e* the type of *e* has to widen to the type of the result variable.

7.2.7 Definite Assignment

The local variables and the result variable of a procedure are not automatically initialised (cf. SCall Rule on p. 133). However, uninitialised variables are a serious threat to type-safe execution of a program. An uninitialised piece of memory may contain an arbitrary sequence of bits. If we regard them as proper values and read them, we can easily produce unpredictable behaviour. Think of an uninitialised pointer variable. The bit sequence is interpreted as a reference to an object in main memory and since the variable is not initialised we may read or write to an arbitrary memory location. For heap allocation we already take special care and initialise the memory with default values (cf. PAlloc Rule on p. 133). For local variables we follow the spirit of Java [42] and supply a simple static analysis for the source program that ensures that we assign a value to a variable before we read from it. For Java this analysis is called "definite assignment" analysis and is already formalised in Isabelle/HOL [102, 59, 58]. We also employ a definite assignment analysis for C0 to protect access to local variables. The analysis does not take global variables into account. We consider that they are already initialised before the program is executed. The formalisation of the definite assignment analysis basically consists of two parts. Function $\mathcal{A}$ calculates the set of variables that are certainly *assigned* to by a piece of code. The test $\mathcal{D}$ that ensures that reading the variable is safe, since it *definitely* was assigned to before.

Definition 7.41 ► *Definite assignment*

The definite assignment analysis for C0 is defined by the functions:

$$\begin{aligned}
\mathcal{D} &:: \textit{'a stmt} \Rightarrow \textit{vname set} \Rightarrow \textit{vname set} \Rightarrow \textit{bool}\\
\mathcal{D}_e &:: \textit{'a expr} \Rightarrow \textit{vname set} \Rightarrow \textit{vname set} \Rightarrow \textit{bool}\\
\mathcal{D}_l &:: \textit{'a expr} \Rightarrow \textit{vname set} \Rightarrow \textit{vname set} \Rightarrow \textit{bool}\\
\mathcal{A} &:: \textit{'a stmt} \Rightarrow \textit{vname set}\\
\mathcal{A}_l &:: \textit{'a expr} \Rightarrow \textit{vname set}
\end{aligned}$$

The definitions are given in Figure 7.14

The functions $\mathcal{D}$ and $\mathcal{A}$ are for statements, $\mathcal{D}_e$ for expressions, and $\mathcal{D}_l$ and $\mathcal{A}_l$ for left-expressions. The parameter *L* is the set of local variables and *A* is the set of assigned variables. The basic test is performed by $\mathcal{D}_e$ (*VarAcc vn T*) *L A*. If variable *vn* is local then it also has to be in *A*. For left expressions the corresponding clause is more liberal. A variable access as left expressions means that we attempt to assign a value to *vn*, which is always allowed. For array access, structure access

$$
\begin{array}{ll}
\mathcal{D}_e\ (\mathit{Lit}\ v\ T)\ L\ A & = \mathit{True}\\
\mathcal{D}_e\ (\mathit{VarAcc}\ vn\ T)\ L\ A & = vn \in L \longrightarrow vn \in A\\
\mathcal{D}_e\ (\mathit{ArrAcc}\ e\ i\ T)\ L\ A & = \mathcal{D}_e\ e\ L\ A \wedge \mathcal{D}_e\ i\ L\ A\\
\mathcal{D}_e\ (\mathit{StructAcc}\ e\ fn\ T)\ L\ A & = \mathcal{D}_e\ e\ L\ A\\
\mathcal{D}_e\ (\mathit{Deref}\ e\ T)\ L\ A & = \mathcal{D}_e\ e\ L\ A\\
\mathcal{D}_e\ (\mathit{BinOp}\ bop\ e_1\ e_2\ T)\ L\ A & = \mathcal{D}_e\ e_1\ L\ A \wedge \mathcal{D}_e\ e_2\ L\ A\\
\mathcal{D}_e\ (\mathit{LazyBinOp}\ bop\ e_1\ e_2\ T)\ L\ A & = \mathcal{D}_e\ e_1\ L\ A \wedge \mathcal{D}_e\ e_2\ L\ A\\
\mathcal{D}_e\ (\mathit{UnOp}\ uop\ e\ T)\ L\ A & = \mathcal{D}_e\ e\ L\ A
\end{array}
\qquad
\begin{array}{ll}
\mathcal{D}_l\ (\mathit{VarAcc}\ vn\ T)\ L\ A & = \mathit{True}\\
\mathcal{D}_l\ (\mathit{ArrAcc}\ e\ i\ T)\ L\ A & = \mathcal{D}_e\ e\ L\ A \wedge \mathcal{D}_e\ i\ L\ A\\
\mathcal{D}_l\ (\mathit{StructAcc}\ e\ fn\ T)\ L\ A & = \mathcal{D}_e\ e\ L\ A\\
\mathcal{D}_l\ (\mathit{Deref}\ e\ T)\ L\ A & = \mathcal{D}_e\ e\ L\ A\\
\mathcal{D}_l\ \text{-}\ L\ A & = \mathit{True}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{A}\ \mathit{Skip} & = \{\}\\
\mathcal{A}\ (\mathit{Ass}\ le\ e) & = \mathcal{A}_l\ le\\
\mathcal{A}\ (\mathit{PAlloc}\ le\ tn) & = \mathcal{A}_l\ le\\
\mathcal{A}\ (\mathit{Comp}\ c_1\ c_2) & = \mathcal{A}\ c_1 \cup \mathcal{A}\ c_2\\
\mathcal{A}\ (\mathit{Ifte}\ b\ c_1\ c_2) & = \mathcal{A}\ c_1 \cap \mathcal{A}\ c_2\\
\mathcal{A}\ (\mathit{Loop}\ b\ c) & = \{\}\\
\mathcal{A}\ (\mathit{SCall}\ vn\ pn\ ps) & = \{vn\}\\
\mathcal{A}\ (\mathit{Return}\ e) & = \{\mathit{Res}\}
\end{array}
\qquad
\begin{array}{ll}
\mathcal{A}_l\ (\mathit{VarAcc}\ vn\ T) & = \{vn\}\\
\mathcal{A}_l\ (\mathit{ArrAcc}\ e\ i\ T) & = \mathcal{A}_l\ e\\
\mathcal{A}_l\ (\mathit{StructAcc}\ e\ fn\ T) & = \mathcal{A}_l\ e\\
\mathcal{A}_l\ (\mathit{Deref}\ e\ T) & = \{\}\\
\mathcal{A}_l\ \text{-} & = \{\}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{D}\ \mathit{Skip}\ L\ A & = \mathit{True}\\
\mathcal{D}\ (\mathit{Ass}\ le\ e)\ L\ A & = \mathcal{D}_l\ le\ L\ A \wedge \mathcal{D}_e\ e\ L\ A\\
\mathcal{D}\ (\mathit{PAlloc}\ le\ tn)\ L\ A & = \mathcal{D}_l\ le\ L\ A\\
\mathcal{D}\ (\mathit{Comp}\ c_1\ c_2)\ L\ A & = \mathcal{D}\ c_1\ L\ A \wedge \mathcal{D}\ c_2\ L\ (A \cup L \cap \mathcal{A}\ c_1)\\
\mathcal{D}\ (\mathit{Ifte}\ b\ c_1\ c_2)\ L\ A & = \mathcal{D}_e\ b\ L\ A \wedge \mathcal{D}\ c_1\ L\ A \wedge \mathcal{D}\ c_2\ L\ A\\
\mathcal{D}\ (\mathit{Loop}\ b\ c)\ L\ A & = \mathcal{D}_e\ b\ L\ A \wedge \mathcal{D}\ c\ L\ A\\
\mathcal{D}\ (\mathit{SCall}\ vn\ pn\ ps)\ L\ A & = \forall e{\in}\mathit{set}\ ps.\ \mathcal{D}_e\ e\ L\ A\\
\mathcal{D}\ (\mathit{Return}\ e)\ L\ A & = \mathit{Res} \in L \wedge \mathcal{D}_e\ e\ L\ A
\end{array}
$$

Figure 7.14: Definite assignment

and dereferencing pointers as left expressions the analysis for ordinary expressions is applied. This means that the first assignment to a structure or array variable must initialise the complete variable at once. Field or index wise initialisation is excluded by this analysis. Function $\mathcal{A}$ collects the variables that are guaranteed to be assigned by the statement. For left-expressions the auxiliary function $\mathcal{A}_l$ descends into the left-expression until it reaches a variable. Global and local variables are collected. For sequential composition *Comp* c_1 c_2 the union of the assigned variables of both statements is returned; for *Ifte b* c_1 c_2 the intersection and for the loop the empty set. This is a safe approximation. For a procedure call the variable assigned to is returned, and *Return e* assigns to the result variable *Res*. For the analysis of the second statement in $\mathcal{D}$ (*Comp* c_1 c_2) $L\ A$ the set A is augmented with the local variables that are assigned to by statement c_1: $L \cap \mathcal{A}\ c_1$.

By induction on the expressions/statement syntax the following basic monotonicity properties of the definite assignment analysis are proven.

If $A \subseteq A'$ *and* $\mathcal{D}_e\ e\ L\ A$ *then* $\mathcal{D}_e\ e\ L\ A'$. ◂ Lemma 7.7

If $A \subseteq A'$ *and* $\mathcal{D}_l\ le\ L\ A$ *then* $\mathcal{D}_l\ le\ L\ A'$. ◂ Lemma 7.8

If $A \subseteq A'$ *and* $\mathcal{D}\ c\ L\ A$ *then* $\mathcal{D}\ c\ L\ A'$. ◂ Lemma 7.9

7.2.8 Wellformed Programs

We consider a program to be wellformed if it respects the following static conditions:

Definition 7.42 ►

$$\begin{array}{l} \textit{wf-prog}\ \Pi \equiv \\ \textsf{let}\ (TD, GD, PD) = \Pi \\ \textsf{in}\ \textit{unique}\ TD \wedge \\ \quad \textit{unique}\ GD \wedge \\ \quad \textit{unique}\ PD \wedge \\ \quad (\forall T \in \textit{snd}\ \text{`}\ \textit{set}\ (TD\ @\ GD).\ \textit{tnenv}\ \Pi \vdash T\ \surd) \wedge (\forall pd \in \textit{set}\ PD.\ \textit{wf-pdefn}\ \Pi\ pd) \end{array}$$

$$\begin{array}{l} \textit{wf-pdefn}\ \Pi\ (pn, (pds, lds, rT), body) \equiv \\ \textit{unique}\ (pds\ @\ lds\ @\ [(Res, rT)]) \wedge \\ (\forall T \in \textit{snd}\ \text{`}\ \textit{set}\ (pds\ @\ lds\ @\ [(Res, rT)]).\ \textit{tnenv}\ \Pi \vdash T\ \surd) \wedge \\ \Pi, \textit{penv}\ \Pi\ (pds, lds, rT), \textit{empty} \vdash \textit{body}\ \surd \wedge \\ \mathcal{D}\ \textit{body}\ (\textit{fst}\ \text{`}\ \textit{set}\ (pds\ @\ lds\ @\ [(Res, rT)]))\ (\textit{fst}\ \text{`}\ \textit{set}\ pds) \wedge Res \in \mathcal{A}\ \textit{body} \end{array}$$

$$\textit{penv}\ \Pi\ (pds, lds, rT) \equiv \textit{map-of}\ (pds\ @\ lds\ @\ [(Res, rT)]\ @\ \textit{gdecls-of}\ \Pi)$$

The type names in the type declarations and the global variables have to be unique and all types have to be wellformed. Moreover, all procedure names have to be unique and the definitions have to be wellformed. The names of parameters, local variables and the result variable *Res* have to be unique and all their types have to be wellformed. The procedure body has to be welltyped with respect to the variable typing obtained from global variables, parameters, local variables and the result variable. This variable typing is obtained with *penv*. Note that the map in *penv* is built from the right to the left. Therefore local names hide global ones. Moreover, the body has to pass the definite assignment test, where parameters, local variables and the result variables are considered as local names and only the parameters are considered as assigned variables. Finally the result variable *Res* has to be assigned in the procedure body.

7.2.9 Type Safety

Type safety relates static semantics like typing and definite assignment with the dynamic semantics, the execution of the program. It describes the properties that are guaranteed during runtime if the static tests have passed. Traditionally [119], type safety is decomposed to *progress* and *subject reduction*. Progress means that the execution cannot get stuck in a configuration were no semantic rule is applicable. This means that the system ran into an undefined or unpredicted situation. Subject reduction means that a typed expression is reduced to a value of the corresponding type. As an example, if evaluation of the branching condition of a loop does not evaluate to a Boolean value but to an integer then the big-step semantics of C0 (cf. Figure 7.7 on p. 133) gets stuck, since neither of the Rules LoopTrue, LoopFalse or LoopFail are applicable.

The C0 big-step semantics in Figure 7.7 is not suited to describe the progress property. It does not distinguish non-termination from stuck computations. Like in the big-step semantics for Simpl (cf. Figure 2.1) one can introduce a special *Stuck* state to signal stuck computations in the final state. Then the semantics does not really get stuck but exits with the *Stuck* state. Hence non-termination and stuck computations can be distinguished. However, the canonical way to prove progress is to use a small-step semantics. Infinite and stuck computations can be distinguished naturally with a small-step semantics. However, since we do not need a progress property for the purpose of the embedding of C0 in Simpl and the corresponding soundness proof, we restrict our attention to subject reduction. We not only prove subject reduction for expressions, but also for statements. That means execution of statements preserves "welltypedness" or *conformance* of the program state.

Conformance of a value to its type is already captured by the typing judgement $\lfloor HT \rfloor \vdash_v v :: T$, where HT is the heap typing for the current state. A *store* like the local variable store or the heap is a mapping from variable names or locations to values, or generally $'a \rightharpoonup val$. A store conforms to static typing $ST :: {'a} \rightharpoonup ty$ if every stored value conforms to its type:

$$HT\vdash s :: ST \equiv \forall p\ v\ T.\ s\ p = \lfloor v \rfloor \longrightarrow ST\ p = \lfloor T \rfloor \longrightarrow \lfloor HT \rfloor \vdash_v v :: T$$ ◂ Definition 7.43

Note that this definition only puts a constraint on positions where both a value and a type are present. It does not demand that for every typed position in the store typing there has to be a value in the store. This is the situation for local variables. They are not initialised when the procedure is entered. However, all global variables have to be initialised. A C0 state is conforming if the heap, the local variables and the global variables conform to the corresponding type environment.

For a program state we define the conformance predicate $TE\vdash s :: HT,LT,GT$, where $TE :: tname \rightharpoonup ty$ is the type environment, $HT :: loc \rightharpoonup tname$ the heap typing, and $LT,GT :: vname \rightharpoonup ty$ the typing for local and global variables, respectively : ◂ Definition 7.44

$$\begin{aligned}&TE\vdash s :: HT,LT,GT \equiv\\&HT\vdash heap\ s :: (TE \circ_m HT) \wedge dom\ (heap\ s) = dom\ HT \wedge finite\ (dom\ (heap\ s)) \wedge\\&HT\vdash lvars\ s :: LT \wedge HT\vdash gvars\ s :: GT \wedge dom\ (gvars\ s) = dom\ GT\end{aligned}$$

The heap typing HT maps locations to type names, and the type environment TE maps type names to types. The heap has to conform to the composition of both. Moreover, the domains of the heap and the heap typing coincide and are finite. The local variables have to conform to their types. And finally the global variables must conform to their types and have to be defined.

We start with some subject reduction theorems for definite assignment. Evaluation of a left-expression preserves the analysis result:

If $leval\ L\ s\ le = \lfloor lv \rfloor$ then $\mathcal{A}_l\ le = \mathcal{A}_l\ lv$. ◂ Lemma 7.10

If $leval\ L\ s\ le = \lfloor lv \rfloor$ and $\mathcal{D}_e\ le\ L'\ A$ then $\mathcal{D}_e\ lv\ L'\ A$. ◂ Lemma 7.11

If $leval\ L\ s\ le = \lfloor lv \rfloor$ and $\mathcal{D}_l\ le\ L'\ A$ then $\mathcal{D}_l\ lv\ L'\ A$. ◂ Lemma 7.12

The proofs are by induction on the left-expression. For statement execution we can prove that the local variables predicted by $\mathcal{A}$ are indeed assigned to by executing the statement:

If $\Pi,L\vdash_{C0} \langle c,\lfloor s \rfloor\rangle \Rightarrow \lfloor t \rfloor$ then $L \cap \mathcal{A}\ c \subseteq dom\ (lvars\ t)$. ◂ Theorem 7.13

Proof. By induction on the big-step execution. □

Function $\mathcal{A}$ returns global and local variables that are assigned to. By intersection with L we only regard the local variables. Note that we do not have to require wellformedness of the program. Since a procedure call restores the local variables of the caller anyway, we do not need to know that all procedure bodies are definitely assigned.

Now we come to the subject reduction theorem for expressions. Evaluation of a welltyped and definitely assigned expression in a conforming state preserves the type:

Theorem 7.14 ► *Subject reduction (expression)*

For a conforming state s: $tnenv\ \Pi \vdash s :: HT, LT{\restriction_A}, GT$, and a welltyped expression e: $\Pi, GT \mathbin{++} LT, HT \vdash_e e\ \surd$ that is definitely assigned: $\mathcal{D}_e\ e\ (dom\ LT)\ A$, we have:
If $eval\ (dom\ LT)\ s\ e = \lfloor v \rfloor$ then $\lfloor HT \rfloor \vdash_v v :: typ\ e$

Proof. By induction on expression *e*. □

The variable environment for the typing of *e* is obtained by overriding the global environment with the local environment: $GT \mathbin{++} LT$. Hence local variables may hide global ones. The type declarations of the program form the type environment. The state only has to be conforming for the local variables in *A*. The definite assignment analysis guarantees that the expression only reads from those variables. The restriction of the local typing *LT* to domain *A* is performed by the restriction operator $LT{\restriction_A}$.

For left expressions we get a similar result:

Theorem 7.15 ► *Subject reduction (left-expression)*

For a conforming state s: $tnenv\ \Pi \vdash s :: HT, LT{\restriction_A}, GT$, and a welltyped left-expression le: $\Pi, GT \mathbin{++} LT, HT \vdash_l le\ \surd$ that is definitely assigned: $\mathcal{D}_l\ le\ (dom\ LT)\ A$, we have:
If $leval\ (dom\ LT)\ s\ le = \lfloor lv \rfloor$ then $\Pi, GT \mathbin{++} LT, HT \vdash_l lv\ \surd$.

Proof. By induction on left-expression *le* and Theorem 7.14. □

Since evaluation of the left-expression again yields a (reduced) left expression the typing judgement for left-expressions is used in the conclusion instead of the value typing in case of ordinary expressions.

To lift subject reduction to the execution of statements we have to be in the context of a wellformed program. This ensures that every procedure is welltyped and definitely assigned. Conformance of the state is preserved by execution:

Theorem 7.16 ► *Subject reduction (statements)*

In context of a wellformed program Π: $wf\text{-}prog\ \Pi$, given a conforming state s: $TE \vdash s :: HT, LT{\restriction_A}, GT$, where $TE = tnenv\ \Pi$ and $GT = genv\ \Pi$, given a statement c that is welltyped: $\Pi, GT \mathbin{++} LT, HT \vdash c\ \surd$ and definitely assigned: $\mathcal{D}\ c\ (dom\ LT)\ A$, then we have:
If $\Pi, dom\ LT \vdash_{C0} \langle c, \lfloor s \rfloor \rangle \Rightarrow \lfloor t \rfloor$ then
$\exists HT'.\ TE \vdash t :: HT', LT{\restriction_{(A \cup \mathcal{A}\ c)}}, GT \wedge HT \subseteq_m HT'$.

Proof. By induction on the big-step execution and Theorems 7.14 and 7.15 and the monotonicity Lemmas 7.9 and 7.7 for definite assignment. □

If the program allocates memory the heap typing has to be extended correspondingly. That is why we obtain an extended heap typing HT' for the final state. Since all the variables in $\mathcal{A}\ c$ are assigned to in the final state, the domain restriction of the local typing can be extended with these variables.

7.3 Conclusion

This chapter presented the formal syntax and semantics of C0, a type-safe subset of the C programming language. The formalisation of C0 is a typical deep embedding of a programming language in HOL, aiming at the meta-theory of C0. The syntax as well as the semantics of all relevant notions of the programming language are defined. The main results are the type safety theorems for C0. They describe the guarantees for the program execution that result form the static welltypedness and definite assignment checks. In particular C0 ensures that all variables and heap locations are initialised and that the execution respects the static typing.

CHAPTER **8**

Embedding C0 into Simpl

This chapter introduces a translation from C0 to Simpl and proves its soundness. Due to this translation, C0 programs can be verified in the verification environment for Simpl and the proven program properties can be transferred to C0 again.

Contents

The program model for C0 presented in the previous chapter is a typical deep embedding of a programming language in HOL. For every relevant notion of the programming language, its types, values, expressions and statements the syntax is defined and then a type system, a definite assignment analysis and an operational semantics is developed on top of the syntax. Finally, properties of the programming language like type safety are proven. The model is well suited for these proofs. We can explicitly reason about welltyped expressions and values, about subject reduction properties and definedness of variables. The type safety proof ensures that for a welltyped and definitely assigned program we do not have to worry about typing issues and definedness of local variables anymore, when investigating individual C0 programs. Therefore it is completely legal to abstract from typing and definedness issues for program verification. The embedding of C0 in Simpl realizes such an abstraction step. C0 variables are identified with record fields, C0 types with HOL types, C0 expressions and atomic statements like assignment or memory allocation are translated to lookup and updates in the state space record.

A peculiarity of the translation of C0 to Simpl is that it cannot be defined generically for all C0 programs, since variables in Simpl are represented as record fields.

Therefore the shape of the record depends on every individual program. However, we can only reason about record fields in HOL as soon as the record-type itself is defined. To remedy this situation we present a two layered approach. We first define and verify a translation of C0 to Simpl that is parametrised on translation functions for access and update of individual variables and heap cells. This proof builds on commutation properties for those basic access and update functions. For each individual program these properties have to proven. By splitting the proof in the parameterised general part and the program specific part it is possible to conduct the main parts once and for all. Moreover, the remaining commutation properties are simple enough to be proven fully automatically for each individual program.

Before going into the details of the formalisation of the abstraction and the corresponding soundness results, we sketch various aspects to give a better intuition.

Workflow In the end we want to prove functional properties and termination of C0 programs. We start with a C0 program, abstract it to Simpl and then verify this Simpl program. The soundness theorem for the abstraction allows to transfer the program properties down to the C0 program again. A Hoare triple specifies the input/output behaviour for all program runs that start in a state satisfying the precondition. To transfer a Hoare triple from Simpl to C0, we need to know that the behaviour of the C0 program is captured by the behaviour of the Simpl program. Then it is also covered by the Hoare triple. Therefore we have to prove that the C0 program can be simulated by the corresponding Simpl program. In context of total correctness we also have to show that termination of the Simpl program implies termination of the corresponding C0 program. The correctness of the C0 embedding in Simpl is not only important to ensure soundness of the overall verification. It is used to transfer proven program properties from a high level of abstraction as in Simpl, to the lower levels of abstraction and in the end to the machine model of the underlying computer hardware. Some parts of the overall system verification can only be carried out on the lower levels. Hence it is pragmatically important to make the proven properties accessible for the lower levels.

Variables Consider a simple C0 program with only two local variables, an integer `i` and a Boolean `b`. In Simpl we represent this state space by a record with two fields:

$$\textbf{record}\ st = i{::}int\ b{::}bool$$

However, in C0 the local variables are represented as a mapping from variable names to values:

$$vname \Rightarrow val\ option$$

By getting rid of the *option* and the *val* layer, the Simpl state space representation introduces two levels of abstraction. All variables are defined and have an individual HOL type. The C0 type system and definite assignment analysis justifies these abstractions.

Variable lookup and assignment is translated to record lookup and update. For example the assignment `i = i + 1` in C0:

Ass (*VarAcc* *"i"* *Integer*)
(*BinOp plus* (*VarAcc* *"i"* *Integer*) (*Lit* (*Prim* (*Intg 1*)) *Integer*) *Integer*),

is abstracted to a *Basic* command in Simpl:

$$\textit{Basic}\ (\lambda s.\ s(\!|i := i\ s + 1|\!))$$

Procedure calls The state-record in Simpl is flat. The local variables of all procedures are side-by-side in the same state-record. A local variable of a procedure is visible within every other procedure. Hence a procedure could read the local variables of another procedure. However, for C0 programs the type system ensures that a procedure only touches its own variables. Pointers to local variables are also excluded in C0. Hence such odd behaviour is ruled out for those Simpl procedures that are translated from a C0 procedure.

Another issue is the procedure call semantics. If a procedure is entered in C0, the local variables are all reset to *None* (cf. SCall Rule in Figure 7.7 on p. 133). In Simpl no such reset takes place. It cannot even be expressed in Simpl, since the record fields yield the plain values without the *option* layer. However, since the definite assignment analysis of C0 ensures that we never access uninitialised variables, we can safely skip resetting the local variables in the corresponding Simpl program.

Heap In C0 the heap is modelled as a mapping from locations to values. Even compound values like structures and arrays fit in one heap cell. The most prominent use of the heap is to store dynamic structures like lists and trees. These are represented as structures in C and also C0, e.g. here is a typical list structure:

```
struct list {
   int cont;
   struct list* next;
}
```

In Simpl we follow the split heap approach introduced in Section 2.4.9.1. The main benefit of splitting the heap is that aliasing between structure fields and hence also aliasing between pointers of different C0 type is already ruled out by the model. To deal with aliasing is the main challenge when reasoning about pointer programs. Reducing the source of potential aliases pays off during program verification.

In the split heap model, we have a separate heap f of type $\mathit{ref} \Rightarrow \mathit{value}$ for each component f of type *value* of the structure. Hence we get:

$$\begin{array}{l}\textbf{record}\ \mathit{heap} = \\ \qquad \mathit{cont} :: \mathit{ref} \Rightarrow \mathit{int} \\ \qquad \mathit{next} :: \mathit{ref} \Rightarrow \mathit{ref}\end{array}$$

In this heap model we obtain individual HOL types for each structure field and most important we rule out aliasing between distinct structure fields. As a consequence, aliasing between different heap structures like lists and trees is ruled out, too. The separation of types and fields in a C0 heap is guaranteed by the C0 type system. In the embedding to Simpl we directly encode it into the heap-model.

To get a uniform representation of structured values in both the heap and the variables, we also split structures in variables. Consider a variable `x` of a pair structure:

```
struct pair {
   int fst;
   int snd;
};

struct pair x;
```

In Simpl we introduce two variables:

record *st* =
  x-fst :: *int*
  x-snd :: *int*

The translation of access and update of C0 structures has to be aware of this different representation. For nested structures, or arrays of structures a kind of normalisation takes place. Consider an array of pairs:

```
struct pair[100] a.
```

Again variable `a` is split up in two variables in Simpl. One array *a-fst* and one array *a-snd*. An array access `a[10].fst` is thus translated to $a\text{-}fst_{[10]}$, since arrays are represented as HOL lists. The array access and the field-selection are swapped in the translation. In general first the field-selections are applied, followed by the array accesses. An expression like `a[10]` yields a pair structure as result. The translation of such an (non-atomic) expression into Simpl depends on the context. For example, if it appears as a parameter of a procedure or in an assignment it is translated to a sequence of updates of the components *a-fst* and *a-snd*.

Allocation The heap in C0 is a mapping from locations to values: *loc* ⇒ *val option*. The locations that store *None* are considered to be free and can be used to allocate a new object. In the split-heap model we have removed the *option* layer. This makes it more convenient to reason about lookup and update in the heap, but we have lost the information about allocated heap locations. We introduce an additional component to the state space, which records the allocation information: a list of allocated locations. In practical applications major parts of the code are only concerned with heap lookup and update and only small parts actually allocate memory. Separating heap and allocation fits well into this scenario.

Runtime Faults Runtime faults in C0 are caused by array bound violations and dereferencing null pointers. In the semantics (left-)expression evaluation and assignment watch for these faults and signal a violation by returning *None*. In Simpl runtime faults are modelled as explicit guards. The expressions themselves do not have to distinguish between *Some* and *None*. Expression evaluation and runtime faults are disentangled in the program logic. This allows to integrate automatic methods that are only concerned with the runtime faults (cf. Section 5) and keep the focus of interactive verification on the functional aspects.

Arithmetic C0 employs bounded modulo arithmetic since it is supposed to run on a real machine. For program verification we want to preserve the opportunity to "think unbounded". To keep consistency with C0 we introduce guards that ensure that the program stays within the arithmetic bounds, and thus no over- or underflows occur during program execution. Strictly speaking the soundness proof for embedding C0 in Simpl is modular with respect to arithmetic. We can plug in the C0 arithmetic, relieving us from guards, or unbounded arithmetic, relieving us from modulo calculations. We keep the flexibility of Simpl to choose the appropriate approach, depending on the application.

Formalisation Issue For the initial example of a C0 program with only two variables, an integer `i` and a Boolean `b`, the abstraction of a variable access to `i` or `b` can be formalised as:

$$\begin{aligned} abs\ (VarAcc\ ''i''\ Integer) &= i \\ abs\ (VarAcc\ ''b''\ Boolean) &= b \end{aligned}$$

We map the variable identifiers to the corresponding record selectors. There are two problems with this abstraction in HOL:

- We cannot generically define how to map a variable identifier to a record field, unless the record is defined.
- The abstraction function *abs* is not welltyped in HOL. The first clause has type $st \Rightarrow int$, the second one $st \Rightarrow bool$.

There is no meta theory of record types in HOL. Only instances of record types start living in a HOL-theory the moment they are defined. Within HOL we cannot generically define how a list of C0 variable declarations is mapped to a record-type with a field for each variable. Hence we can only define the translation from C0 to Simpl for each individual C0 program, since we then know the state space record. However, fortunately we do not have to redo the complete soundness proof for each C0 program. Major parts of the translation from C0 to Simpl can indeed be formulated in HOL. Only for the basic operations concerning access/update of a variable or structure field we need to know the state record. We can use these basic actions as parameters to the abstraction function as well as the theorems and proofs. For example, we can supply a lookup function V to the abstraction function, which tells how a lookup of a C0 variable works in the Simpl state space:

$$abs\ V\ (VarAcc\ vn\ T)\ =\ V\ vn$$

This generalised abstraction function can be defined generically for all C0 programs. The generic soundness proof puts constraints on the lookup function V. For each individual C0 program we define V and have to prove these constraints. These proof obligations are simple enough to be discharged automatically. For variable updates we employ the same idea.

For the same typing issues mentioned before we cannot simultaneously define $V\ ''i'' = i$ and $V\ ''b'' = b$. To remedy the situation we fall back to the C0 values:

$$\begin{aligned} V\ ''i'' &= \lambda s.\ Prim\ (Intg\ (i\ s)) \\ V\ ''b'' &= \lambda s.\ Prim\ (Bool\ (b\ s)) \end{aligned}$$

This gives raise to the question if we have lost one of the major goals of the embedding into Simpl: to get rid of the explicit typing in terms and use HOL types instead. Fortunately the answer is no. Since the expressions are always embedded in statements, their values are only intermediate results in a state update. Hence in the end *val* constructors and destructors cancel each other. For example, the abstraction function translates the assignment `i = i + 1` to

$$Basic\ (\lambda s.\ s(\!|i := the\text{-}Intg_v\ (Prim\ (Intg\ (i\ s))) + the\text{-}Intg_v\ (Prim\ (Intg\ 1))|\!)).$$

This rewrites to the desired

$$Basic\ (\lambda s.\ s(\!|i := i\ s + 1|\!)).$$

Since we only consider welltyped C0 programs this rewriting step always works.

8.1 Splitting Types and Values

Basically C0 values are abstracted to Simpl by getting rid of the value constructors and by splitting structures to their components.

C0	Simpl
Prim (*Bool b*)	*b*
Prim (*Intg i*)	*i*
Prim (*Unsgnd n*)	*n*
Prim (*Chr c*)	*c*
Prim Null	*NULL*
Prim (*Addr l*)	*Rep-loc l*
Structv fs	split *map snd fs*
Arrv vs	split *vs*

Value *Null* is translated to reference *NULL* and addresses are converted to references by the representation function *Rep-loc*. Since locations are defined as non *NULL* references (cf. p. 122) we always get a proper reference. Structures are (recursively) split to their primitive values. For example, for a pair structure like *Structv* [(*"fst"*, *Prim* (*Intg i*)), (*"snd"*, *Prim* (*Intg j*))] the components *i* and and *j* are stored separately in the Simpl state. Arrays are represented as a list of primitive values. We cannot directly define a HOL function that performs the transformation described in the table, since its result type depends on the input value. For example, for a *Bool b* a Boolean is returned but for *Intg i* an integer. As motivated in the previous section values only appear nested in statements and there we can apply the corresponding value destructors. Moreover, we can reason about splitting values to their atomic components without leaving *val*. The essence of the abstraction to Simpl is expressed in terms of the concrete C0 representation. This idea is used throughout the whole rest of this chapter. We describe the effect of abstract operations in the Simpl level by the corresponding effect in the C0 level. Because of the deep embedding of C0 types and values it is straightforward to define manipulations on this level.

We call a C0 type or value *atomic* if we do not have to split it for the embedding into Simpl. All primitive values are atomic, arrays are atomic if the element type is atomic and structures are atomic only in the borderline case when they do not have any fields:

Definition 8.1 ►

$$\begin{array}{ll} atomic_T :: ty \Rightarrow bool & \\ atomic_T\ (Struct\ fTs) & = fTs = [] \\ atomic_T\ (Arr\ n\ T) & = atomic_T\ T \\ atomic_T\ \text{-} & = True \end{array}$$

As running examples we introduce the structures `nested` and `pair`:

```
struct nested {                        struct pair {
   struct { int x; bool y;} a;            int fst;
   int b;                                 int snd;
}                                      }
```

The function *selectors* collects all field selectors of a type and returns them as a list of paths to the atomic components. For example, for `nested` we get:

$$[["a", "x"], ["a", "y"], ["b"]]$$

For arrays the selectors of the element type are returned.

◂ Definition 8.2

$$\begin{aligned}
&\mathit{selectors} :: \mathit{ty} \Rightarrow \mathit{fname\ list\ list}\\
&\mathit{selectors}\ (\mathit{Struct\ fTs}) = \textsf{if}\ \mathit{fTs} = []\ \textsf{then}\ [[]]\\
&\qquad \textsf{else}\ \mathit{concat}\ (\mathit{map}\ (\lambda(f, T).\ \mathit{map}\ ((\cdot)\ f)\ (\mathit{selectors}\ T))\ \mathit{fTs})\\
&\mathit{selectors}\ (\mathit{Arr}\ n\ T) = \mathit{selectors}\ T\\
&\mathit{selectors}\ \text{-} = [[]]
\end{aligned}$$

Given a type T and a selector path ss, function sel_T (T, ss) retrieves the selected component type $\lfloor sT \rfloor$ if the path is valid and *None* otherwise. The selection distributes over array types, e.g. given an array of pairs, the path [*"fst"*] selects an array of integers.

◂ Definition 8.3

$$\begin{aligned}
&sel_T :: \mathit{ty} \times \mathit{fname\ list} \Rightarrow \mathit{ty\ option}\\
&sel_T\ (\mathit{Boolean}, []) = \lfloor \mathit{Boolean} \rfloor\\
&sel_T\ (\mathit{Integer}, []) = \lfloor \mathit{Integer} \rfloor\\
&sel_T\ (\mathit{UnsgndT}, []) = \lfloor \mathit{UnsgndT} \rfloor\\
&sel_T\ (\mathit{CharT}, []) = \lfloor \mathit{CharT} \rfloor\\
&sel_T\ (\mathit{NullT}, []) = \lfloor \mathit{NullT} \rfloor\\
&sel_T\ (\mathit{Ptr\ tn}, []) = \lfloor \mathit{Ptr\ tn} \rfloor\\
&sel_T\ (\mathit{Struct\ fTs}, []) = \lfloor \mathit{Struct\ fTs} \rfloor\\
&sel_T\ (\mathit{Struct\ fTs}, s{\cdot}ss) = \textsf{case}\ \mathit{map\text{-}of\ fTs\ s}\ \textsf{of}\ \mathit{None} \Rightarrow \mathit{None} \mid \lfloor T \rfloor \Rightarrow sel_T\ (T, ss)\\
&sel_T\ (\mathit{Arr}\ n\ T, ss) = \textsf{case}\ sel_T\ (T, ss)\ \textsf{of}\ \mathit{None} \Rightarrow \mathit{None} \mid \lfloor eT \rfloor \Rightarrow \lfloor \mathit{Arr}\ n\ eT \rfloor\\
&sel_T\ (\text{-},\text{-}) = \mathit{None}
\end{aligned}$$

We can relate *selectors* and sel_T. If we can select an atomic type, then the path is among the selectors:

◂ Lemma 8.1

If $sel_T\ (T, ss) = \lfloor sT \rfloor$ *and* $atomic_T\ sT$ *then* $ss \in set\ (selectors\ T)$.

Proof. By induction along the recursion-scheme of sel_T. □

For the opposite direction we need to know that the selectors are distinct. Otherwise a selector could be hidden by another one with the same name, since the *map-of* used by sel_T only finds the most recent field.

◂ Definition 8.4

$$\begin{aligned}
&\mathit{distinct\text{-}selectors} :: \mathit{ty} \Rightarrow \mathit{bool}\\
&\mathit{distinct\text{-}selectors}\ (\mathit{Struct\ fTs}) = \mathit{distinct}\ (\mathit{map\ fst\ fTs}) \wedge\\
&\qquad (\forall T{\in}\mathit{snd}\ `\ \mathit{set\ fTs}.\ \mathit{distinct\text{-}selectors}\ T)\\
&\mathit{distinct\text{-}selectors}\ (\mathit{Arr}\ n\ T) = \mathit{distinct\text{-}selectors}\ T\\
&\mathit{distinct\text{-}selectors}\ \text{-} = \mathit{True}
\end{aligned}$$

Lemma 8.2 ► *If distinct-selectors T* and *ss* $\in$ *set (selectors T)* then
$\exists sT.\ sel_T\ (T, ss) = \lfloor sT \rfloor \wedge atomic_T\ sT$.

Proof. By induction along the recursion-scheme of *selectors*. □

Every wellformed type has distinct selectors.

Lemma 8.3 ► *If* $TE \vdash T \surd$ then *distinct-selectors T*.

Proof. By induction on wellformedness of types. □

Similar to sel_T we define a function idx_T to index array types. It selects the element type of an array or sub-array types of multi-dimensional arrays. It gets a list of indices as parameter. Only the length of this list is relevant.

Definition 8.5 ►

$$\begin{aligned}
&idx_T :: ty \times 'a\ list \Rightarrow ty\ option\\
&idx_T\ (T, [\,]) = \lfloor T \rfloor\\
&idx_T\ (Arr\ n\ T, i{\cdot}is) = idx_T\ (T, is)\\
&idx_T\ (\text{-},\text{-}) = None
\end{aligned}$$

A central property of splitting types is that selections can always be applied before indexing. This reflects the essence of the split-heap approach. Selection means to pick a heap, which always is the first step.

Lemma 8.4 ► Let $sel_T\ (T, ss) = \lfloor sT \rfloor$ and $idx_T\ (sT, is) = \lfloor iT \rfloor$ and $sel_T\ (iT, ss') = \lfloor siT \rfloor$. Then
$\exists ssT.\ sel_T\ (T, ss\ @\ ss') = \lfloor ssT \rfloor \wedge idx_T\ (ssT, is) = \lfloor siT \rfloor$.

Proof. By induction on the recursion-scheme of $sel_T\ (T, ss)$. □

The functions sel_T and idx_T yield an option type. They check whether a selector path or index list fits to a type. Given a value of the same type the functions sel_v and idx_v select the corresponding sub-value.

Definition 8.6 ►

$$\begin{aligned}
&sel_v :: val \times fname\ list \Rightarrow val\\
&sel_v\ (Prim\ v, [\,]) = Prim\ v\\
&sel_v\ (Structv\ fVs, [\,]) = Structv\ fVs\\
&sel_v\ (Structv\ fVs, s{\cdot}ss) = \textsf{case}\ map\text{-}of\ fVs\ s\ \textsf{of}\ None \Rightarrow arbitrary \mid \lfloor v \rfloor \Rightarrow sel_v\ (v, ss)\\
&sel_v\ (Arrv\ vs, ss) = Arrv\ (map\ (\lambda v.\ sel_v\ (v, ss))\ vs)
\end{aligned}$$

Definition 8.7 ►

$$\begin{aligned}
&idx_v :: val \times nat\ list \Rightarrow val\\
&idx_v\ (Prim\ x, [\,]) = Prim\ x\\
&idx_v\ (Arrv\ vs, i{\cdot}is) = idx_v\ (vs_{[i]}, is)
\end{aligned}$$

The predicate *dimfits* tests, whether the dimension (length) of an index list fits to a type:

Definition 8.8 ►

$$\begin{aligned}
&dimfits :: ty \Rightarrow 'a\ list \Rightarrow bool\\
&dimfits\ T\ is = (\exists iT.\ idx_T\ (T, is) = \lfloor iT \rfloor)
\end{aligned}$$

To safely index a value we also have to ensure that every index lies within the bounds of the array:

Definition 8.9 ►

$$\begin{aligned}
&idxfits :: (val \times nat\ list) \Rightarrow bool\\
&idxfits\ (v, [\,]) = True\\
&idxfits\ (Arrv\ av, i{\cdot}is) = i < |av| \wedge idxfits\ (av_{[i]}, is)\\
&idxfits\ (\text{-},\text{-}) = False
\end{aligned}$$

A central property of selecting sub-values with sel_v is the preservation of the array structure. For example, if we take an array A of pairs with *100* elements then both the array $sel_v\ (A,\ [''fst''])$ and $sel_v\ (A,\ [''snd''])$ have *100* elements. This carries over to multiple dimensions and selections:

If $\vdash_v v :: T$ *and* $sel_T\ (T,\ ss) = \lfloor sT \rfloor$ *and* $idxfits\ (v,\ is)$ *then* $idxfits\ (sel_v\ (v,\ ss),\ is)$. ◂ Lemma 8.5

Proof. By induction on the recursion-scheme of sel_T. □

In Simpl we employ the split-heap model. Every value is split into its atomic components. The function *explode* describes this effect. It takes a type and a value and returns the list of atomic values according to the selectors of the type.

◂ Definition 8.10

$$\begin{array}{l} explode :: (ty \times val) \Rightarrow val\ list \\ explode\ (T,\ v) \equiv map\ (\lambda ss.\ sel_v\ (v,\ ss))\ (selectors\ T) \end{array}$$

For the reverse effect, to build a compound value out of the list of atomic values, we use the type as blueprint for the value.

◂ Definition 8.11

$$\begin{array}{ll} \multicolumn{2}{l}{implode :: (ty \times val\ list) \Rightarrow val} \\ implode\ (Boolean,\ [b]) & = b \\ implode\ (Integer,\ [i]) & = i \\ implode\ (UnsgndT,\ [n]) & = n \\ implode\ (CharT,\ [c]) & = c \\ implode\ (Ptr\ tn,\ [p]) & = p \\ implode\ (NullT,\ [p]) & = p \\ implode\ (Struct\ fTs,\ vs) & = \mathsf{let}\ fs = map\ fst\ fTs; \\ & \qquad Ts = map\ snd\ fTs; \\ & \qquad vss = partition\ vs\ (map\ (length \circ selectors)\ Ts); \\ & \qquad vs' = map\ implode\ (zip\ Ts\ vss) \\ & \quad \mathsf{in}\ Structv\ (zip\ fs\ vs') \\ implode\ (Arr\ n\ T,\ vs) & = Arrv \\ & \quad (map\ (\lambda vs.\ implode\ (T,\ vs))\ (transpose\ (map\ the\text{-}Arrv\ vs))) \end{array}$$

For primitive values the list of atomic values is the singleton list.

For structures the list of values is first partitioned to the sublists corresponding to the selectors of each field. These sublists are imploded.

◂ Definition 8.12

$$\begin{array}{ll} \multicolumn{2}{l}{partition :: 'a\ list \Rightarrow nat\ list \Rightarrow 'a\ list\ list} \\ partition\ xs\ [] & = [] \\ partition\ xs\ (n \cdot ns) & = take\ n\ xs \cdot partition\ (drop\ n\ xs)\ ns \end{array}$$

In the `nested` example the split version of a value is of the form

$$[Prim\ (Intg\ x),\ Prim\ (Bool\ y),\ Prim\ (Intg\ b)].$$

The first two values belong to sub-structure *a*. Hence partitioning groups the first two elements together:

$$[[Prim\ (Intg\ x),\ Prim\ (Bool\ y)],\ [Prim\ (Intg\ b)]].$$

Since selection distributes over arrays, the value lists have to be transposed before they are imploded.

Definition 8.13 ►

$$
\begin{aligned}
&\textit{transpose} :: \textit{'a list list} \Rightarrow \textit{'a list list}\\
&\textit{transpose}\ [] &&= []\\
&\textit{transpose}\ ([]\cdot xss) &&= \textit{transpose}\ xss\\
&\textit{transpose}\ ((x\cdot xs)\cdot xss) &&= \textit{map hd}\ ((x\cdot xs)\cdot xss)\cdot \textit{transpose}\ (\textit{map tl}\ ((x\cdot xs)\cdot xss))
\end{aligned}
$$

For example, an array of `pairs` is exploded into two lists, one with the first elements and one with the second elements. Before imploding it, we group the corresponding first and second elements together by transposing the matrix of elements:

$$\textit{transpose}\ [[\textit{fst}_1, \textit{fst}_2, \textit{fst}_3], [\textit{snd}_1, \textit{snd}_2, \textit{snd}_3]] = [[\textit{fst}_1, \textit{snd}_1], [\textit{fst}_2, \textit{snd}_2], [\textit{fst}_3, \textit{snd}_3]].$$

If a value v has type T, and T has distinct selectors, then imploding reverts the effect of exploding v. This reflects the situation between corresponding values in a C0 state and a Simpl state. In Simpl the value is stored in its exploded version.

Theorem 8.6 ► *If* $\vdash_v v :: T$ *and distinct-selectors* T *then* $\textit{implode}\ (T, \textit{explode}\ (T, v)) = v$.

Proof. By induction on $\vdash_v v :: T$. □

8.2 Expressions

An expression is evaluated in a certain program state. Its value can depend on the state of global and local variables and the heap. The shape of the Simpl state record is not known until we have a concrete C0 program to translate. Since we want to define a generic abstraction from C0 to Simpl, we supply lookup functions for variables and heap as parameters. These lookup functions have to get all the necessary information, so that we can properly implement them later for individual C0 programs. The construction of these functions is completely schematic and can thus be automatised (cf. Section 8.8). We start with global variables. At least we have to know the C0 variable name *vn*. Since we split all compound variables to their atomic components we also supply a selector path *ss*. To properly deal with multi-dimensional arrays we also consider an index list *is*. For example, a two dimensional array of integers is stored as *int list list* in the Simpl state-record. If an empty index list is supplied, this two dimensional list is converted to a C0 array value. If one index is given we first select the list in the Simpl state and get an *int list* that is converted to a C0 array value. If two indexes are given we first select the element and then make a primitive C0 value out of it. This strategy avoids to introduce too many C0 value constructors in the intermediate values. Otherwise it may happen that C0 value constructors are still visible in the resulting Simpl program, which we want to avoid. We come back to this point when the abstraction function for expressions is defined. For example, consider a global array of pairs: `struct pair [100] a`. To lookup a component we supply the name *"a"*, a selector path [], [*"fst"*] or [*"snd"*] and a proper index list [] or [*i*].

Altogether we have the following signature of the lookup function for global variables, where *'s* is the type variable for the Simpl state:

$$GV :: \textit{vname} \Rightarrow \textit{fname list} \Rightarrow \textit{'s} \Rightarrow \textit{nat list} \Rightarrow \textit{val}.$$

Some remarks on the order of parameters. The variable name in Simpl can depend on the C0 variable name and the selector path. So $GV\ vn\ ss\ s_a$ denotes the Simpl representation of the variable (component), where GV is the name of the lookup function, and s_a is a Simpl state. An atomic value in Simpl can be a primitive value or a multi-dimensional list. This is modelled by the type: *nat list* ⇒ *val*.

To lookup local variables we additionally supply the current procedure name *pn* to disambiguate name collisions:

$$LV :: pname \Rightarrow vname \Rightarrow fname\ list \Rightarrow {}'s \Rightarrow nat\ list \Rightarrow val.$$

For heap lookup we identify the (split-)heap in Simpl by the type-name and a selector path. The address is identified by the reference:

$$H :: tname \Rightarrow fname\ list \Rightarrow ref \Rightarrow {}'s \Rightarrow nat\ list \Rightarrow val.$$

The following definitions and theorems all depend on these lookup functions. They are common parameters in the abstraction functions and theorems. To group such common parameters and even specify them by a set of assumptions, Isabelle provides a mechanism called **locale** [10]. These locales can be used to realise modular reasoning. First an abstract theory in the context of the locale is developed. Later on, one can instantiate the locale parameters and provide proofs for the locale assumptions. Then Isabelle automatically specialises all the locale theorems to the concrete application. This exactly fits to our situation. Currently we develop an abstract theory, how to embed C0 to Simpl and prove crucial properties of this translation. We collect the necessary requirements on the basic lookup and update functions. When we later on verify individual C0 programs, we supply these basic lookup and update functions, prove their requirements and finally can use the soundness theorem for the translation.

The first locale *expr* defines the context in which we define the abstraction of expressions from C0 to Simpl. To highlight which definitions and theorems depend on a locale they are marked with (in <*locale-name*>).

◂ Definition 8.14

locale *expr* =
fixes
global variable lookup:
$GV :: vname \Rightarrow fname\ list \Rightarrow {}'s \Rightarrow nat\ list \Rightarrow val$
local variable lookup:
$LV :: pname \Rightarrow vname \Rightarrow fname\ list \Rightarrow {}'s \Rightarrow nat\ list \Rightarrow val$
heap variable lookup:
$H :: tname \Rightarrow fname\ list \Rightarrow ref \Rightarrow {}'s \Rightarrow nat\ list \Rightarrow val$
unary operations:
$U :: (unop \times prim) \Rightarrow prim$
binary operations:
$B :: (binop \times prim \times prim) \Rightarrow prim$

defines
variable lookup:
$V :: vname\ set \Rightarrow pname \Rightarrow vname \Rightarrow fname\ list \Rightarrow {}'s \Rightarrow nat\ list \Rightarrow val$
$V\ L\ pn\ vn\ ss\ s_a\ is \equiv \textsf{if}\ vn \in L\ \textsf{then}\ LV\ pn\ vn\ ss\ s_a\ is\ \textsf{else}\ GV\ vn\ ss\ s_a\ is$

GV, *LV* and *H* are the lookup functions for global and local variables and the heap. The locale also allows to give local definitions of parameters, like the variable lookup function V. It gets a set of local variables L as parameter and decides whether to make a local lookup via LV or a global lookup via GV. As naming convention the subscript a in state s_a is used for abstract Simpl states and a plain s is used for C0 states. The functions U and B are used to abstract the unary and binary operations of C0. As mentioned in the overview in the previous section this allows us to postpone the decision how to reason about arithmetic in the program logic. We also parametrise unary operations, because of the type-cast *to-int*. In C0 it is also defined with modulo arithmetic (cf. Figure 7.4 on p. 130). If we use unbounded arithmetic in the Hoare logics the HOL conversion *int* can be used instead.

In Simpl there are no explicit syntactic terms for expressions. Expressions are shallow-embedded as functions depending on the current state. Hence abstracting a C0 expression to Simpl basically means to evaluate a C0 expression in an abstract Simpl state.

Definition 8.15 ► (in *expr*) *Abstracting expressions*

Abstraction of expressions has the format:

$$abs_e\ L\ pn\ ss\ is\ e\ s_a$$

and is defined in Figure 8.1, where L is a set of local variables, pn a procedure name, ss a selector path, is an index list, e an expression and s_a an abstract Simpl state.

$abs_e :: vname\ set \Rightarrow pname \Rightarrow fname\ list \Rightarrow nat\ list \Rightarrow ty\ expr \Rightarrow {'s} \Rightarrow val$

$abs_e\ L\ pn\ ss\ is\ (Lit\ v\ T)\ s_a = idx_v\ (sel_v\ (v, ss), is)$

$abs_e\ L\ pn\ ss\ is\ (VarAcc\ vn\ T)\ s_a = V\ L\ pn\ vn\ ss\ s_a\ is$

$abs_e\ L\ pn\ ss\ is\ (ArrAcc\ e\ i\ T)\ s_a =$
$\textsf{let}\ n = the\text{-}Unsgnd_v\ (abs_e\ L\ pn\ []\ []\ i\ s_a)\ \textsf{in}\ abs_e\ L\ pn\ ss\ (n{\cdot}is)\ e\ s_a$

$abs_e\ L\ pn\ ss\ is\ (StructAcc\ e\ fn\ T)\ s_a = abs_e\ L\ pn\ (fn{\cdot}ss)\ is\ e\ s_a$

$abs_e\ L\ pn\ ss\ is\ (Deref\ e\ T)\ s_a =$
$\textsf{let}\ r = the\text{-}Ref\ (abs_e\ L\ pn\ []\ []\ e\ s_a);$
$\quad tn = tname\ (typ\ e)$
$\textsf{in}\ H\ tn\ ss\ r\ s_a\ is$

$abs_e\ L\ pn\ ss\ is\ (UnOp\ uop\ e\ T)\ s_a = Prim\ (U\ (uop, the\text{-}Prim\ (abs_e\ L\ pn\ ss\ is\ e\ s_a)))$

$abs_e\ L\ pn\ ss\ is\ (BinOp\ bop\ e_1\ e_2\ T)\ s_a =$
$Prim\ (B\ (bop, the\text{-}Prim\ (abs_e\ L\ pn\ ss\ is\ e_1\ s_a), the\text{-}Prim\ (abs_e\ L\ pn\ ss\ is\ e_2\ s_a)))$

$abs_e\ L\ pn\ ss\ is\ (LazyBinOp\ bop\ e_1\ e_2\ T)\ s_a =$
$Prim\ (the\ (apply\text{-}lazybinop\ (bop, the\text{-}Prim\ (abs_e\ L\ pn\ ss\ is\ e_1\ s_a), the\text{-}Prim\ (abs_e\ L\ pn\ ss\ is\ e_2\ s_a))))$

Figure 8.1: Abstraction of expressions

The parameters ss for the selector path, and is for the index list are accumulators. On the top-level $abs_e\ L\ pn\ []\ []\ e\ s_a$ is invoked. As abs_e descends into the expression it augments ss with a selector when it passed a structure access, and is with an index as it passes an array access. The local variables L and the procedure name pn specify the current procedure context of the expression.

For literal values the functions idx_v and sel_v are used to select the sub-value according to the accumulated *ss* and *is*. This clause also reflects the essence of splitting values, variables and heaps in Simpl. Structure and array accesses are normalised. First all structure accesses are applied by sel_v followed by the array indexing. In the corresponding C0 expression, selection and indexing can be interleaved, depending on the type. For example, in an array of pairs we first have to index the array before we can select the first component of a pair. In Simpl there are two arrays, one storing all the first components and one storing the second components. By choosing the array to index we have already implicitly applied the selection.

Variable access is handled by lookup function V.

For array access *ArrAcc e i T* the index is evaluated and added in front of *is* before descending into *e*.

Analogous for structure access *StructAcc e fn T* the field name is augmented to *ss* before descending into *e*.

For dereferencing pointers *Deref e T* we calculate the reference and extract the type-name from the type of *e* and finally use the lookup function H. The auxiliary function *tname* is defined as follows:

$$\begin{array}{l} tname :: ty \Rightarrow tname \\ tname\ (Ptr\ tn) = tn \end{array}$$

◂ Definition 8.16

For unary, binary and lazy binary expressions first the subexpressions are evaluated. The result is passed to U in case of an unary operation, to B in case of a binary operation, or otherwise directly to *apply-lazybinop*.

For the abstraction abs_e, the proper values for *ss* and *is* depend on the type of expression *e*. For primitive types both *ss* and *is* have to be empty, since neither selection nor indexing of primitive types makes any sense. Since all unary, binary and lazy-binary operations work on values of primitive types we can always consider *ss* and *is* to be empty.

The selector path is propagated through the term, since all variables and heap-components are split in Simpl. Hence we collect all the selectors in order to pick the corresponding component as we finally reach a variable or heap access.

It might be puzzling to propagate the indexes through the term. It may be more intuitive to omit the index accumulator *is* completely, and simply generate a list lookup whenever an array access *ArrAcc e i T* is processed:

$$\begin{array}{l} abs'\ L\ pn\ ss\ (ArrAcc\ e\ i\ T)\ s_a = \\ \textsf{let}\ a = the\text{-}Arrv\ (abs'\ L\ pn\ ss\ e\ s_a); \\ \quad n = the\text{-}Unsgnd_v\ (abs'\ L\ pn\ []\ i\ s_a) \\ \textsf{in}\ a_{[n]} \end{array}$$

The problem becomes apparent by the following example. Consider a global array `arr` of integers , and an array access `Arr[i]`. Moreover, let this array be mapped to a record field *arr* of type *int list* in the Simpl state space. The desired Simpl expression for array indexing is the following:

$$(arr\ s_a)_{[i\ s_a]}$$

However, if we follow *abs'* we get a different term. The abstraction of e yields

$$Arrv\ (map\ (Prim \circ Intg)\ (arr\ s_a)),$$

since it transforms the *int list* to a *val*. Only the constructor *Arrv* is cancelled by the destructor *the-Arrv*. Hence we end up with:

$$\textit{the-Intg}_v\ (\textit{map}\ (\textit{Prim} \circ \textit{Intg})\ (\textit{arr}\ s_a))_{[i\ s_a]}.$$

Unfortunately we cannot distribute the destructor *the-Intg*$_v$ under the list selection. This only works if we know that the index is in range of the list: $i\ s_a < |arr\ s_a|$. Even if this information is present in a guard in the C0 program, we need semantic arguments to justify that the guard in front of the expression holds and thus the destructor can be distributed. Mere syntactic equivalence of both Simpl programs is not valid. We can completely avoid this problem if we do not construct the complete array as C0 value in the first place, but just the primitive value resulting from list selection. Exactly this is implement]ed by propagating the array indexes to the core lookup functions. The lookup function *GV* can take care of this:

$$\begin{array}{l} \textit{GV}\ \textit{"arr"}\ []\ s_a\ \textit{is} = \\ \textsf{case}\ \textit{is}\ \textsf{of}\ [] \Rightarrow \textit{Arrv}\ (\textit{map}\ (\textit{Prim} \circ \textit{Intg})\ (\textit{arr}\ s_a)) \\ \mid i{\cdot}\textit{is} \Rightarrow \textit{Prim}\ (\textit{Intg}\ (\textit{arr}\ s_a)_{[i]}) \end{array}$$

The abstraction *abs*$_e$ does not care about potential runtime faults, like dereferencing a null pointer or array bound violations. To properly simulate C0 expressions in Simpl we additionally generate guards that watch for these runtime faults. Moreover, we may have to introduce guards for unary and binary operations. If we use unbounded arithmetic in Simpl we have to introduce guards that prevent over- and underflows.

Guards are modelled as state sets in Simpl. As an optimisation we introduce a guard generating function *guard*$_e$ that produces optional state sets. The result *None* means that no guard is necessary. Semantically *None* can be viewed as syntactic variant of the universal state set. The "guard" *None* always holds and thus we can omit it.

Definition 8.17 ► *Figure 8.2 defines the operations $\sqcup$, $\sqcap$, $\in\in$ and $\sqsubseteq$. They extend operations $\cup$, $\cap$, $\in$ and $\subseteq$ on 'a set to 'a set option, treating None as the universal set.*

$$\boxed{\begin{array}{lcl} A \sqcup B & \equiv & \textsf{case}\ A\ \textsf{of}\ \textit{None} \Rightarrow \textit{None} \mid \lfloor A \rfloor \Rightarrow \textsf{case}\ B\ \textsf{of}\ \textit{None} \Rightarrow \textit{None} \mid \lfloor B \rfloor \Rightarrow \lfloor A \cup B \rfloor \\ A \sqcap B & \equiv & \textsf{case}\ A\ \textsf{of}\ \textit{None} \Rightarrow B \mid \lfloor A \rfloor \Rightarrow \textsf{case}\ B\ \textsf{of}\ \textit{None} \Rightarrow \lfloor A \rfloor \mid \lfloor B \rfloor \Rightarrow \lfloor A \cap B \rfloor \\ a \in\in A & \equiv & \textsf{case}\ A\ \textsf{of}\ \textit{None} \Rightarrow \textit{True} \mid \lfloor A \rfloor \Rightarrow a \in A \\ A \sqsubseteq B & \equiv & \textsf{case}\ B\ \textsf{of}\ \textit{None} \Rightarrow \textit{True} \mid \lfloor B \rfloor \Rightarrow \textsf{case}\ A\ \textsf{of}\ \textit{None} \Rightarrow \textit{False} \mid \lfloor A \rfloor \Rightarrow A \subseteq B \end{array}}$$

Figure 8.2: Operations on *'a set option*

To guard unary and binary operations we extend locale *expr* with the guard generating functions U_g and B_g.

Definition 8.18 ► **locale** *guard* = *expr* +
fixes
guard for unary operations:
$U_g :: \textit{unop} \times \textit{ty} \Rightarrow (\textit{'s} \Rightarrow \textit{prim}) \Rightarrow \textit{'s set option}$
guard for binary operations:
$B_g :: \textit{binop} \times \textit{ty} \times \textit{ty} \Rightarrow (\textit{'s} \Rightarrow \textit{prim}) \Rightarrow (\textit{'s} \Rightarrow \textit{prim}) \Rightarrow \textit{'s set option}$

The arguments of type $'s \Rightarrow prim$ describe the values of the parameters for the unary or binary operation. The abstraction on the state $'s$ is needed, since the value can depend on the state and may have to be incorporated into the resulting guard, which is a state set and thus also depends on the current state.

◂ Definition 8.19 (in *guard*) *Guarding expressions*

Figure 8.3 defines the guard generation for expression e:

$$guard_e\ L\ pn\ ss\ is\ e,$$

where L is a set of local variables, pn a procedure name, ss a selector path, is an index list, e an expression and s_a an abstract Simpl state.

$guard_e :: vname\ set \Rightarrow pname \Rightarrow fname\ list \Rightarrow ('s \Rightarrow nat)\ list \Rightarrow ty\ expr \Rightarrow 's\ set\ option$

$guard_e\ L\ pn\ ss\ is\ (Lit\ v\ T) = guard_i\ (\lambda s_a\ is.\ idx_v\ (sel_v\ (v, ss), is))\ []\ is$

$guard_e\ L\ pn\ ss\ is\ (VarAcc\ vn\ T) = guard_i\ (V\ L\ pn\ vn\ ss)\ []\ is$

$guard_e\ L\ pn\ ss\ is\ (ArrAcc\ e\ i\ T) =$
$\mathsf{let}\ n = \lambda s_a.\ the\text{-}Unsgnd_v\ (abs_e\ L\ pn\ []\ []\ i\ s_a)\ \mathsf{in}\ guard_e\ L\ pn\ ss\ (n{\cdot}is)\ e \sqcap guard_e\ L\ pn\ []\ []\ i$

$guard_e\ L\ pn\ ss\ is\ (StructAcc\ e\ fn\ T) = guard_e\ L\ pn\ (fn{\cdot}ss)\ is\ e$

$guard_e\ L\ pn\ ss\ is\ (Deref\ e\ T) =$
$\mathsf{let}\ r = \lambda s_a.\ the\text{-}Ref\ (abs_e\ L\ pn\ []\ []\ e\ s_a);$
$\quad tn = tname\ (typ\ e)$
$\mathsf{in}\ guard_e\ L\ pn\ []\ []\ e \sqcap \lfloor\{s_a.\ the\text{-}Ref\ (abs_e\ L\ pn\ []\ []\ e\ s_a) \neq NULL\}\rfloor \sqcap$
$\quad guard_i\ (\lambda s_a.\ H\ tn\ ss\ (r\ s_a)\ s_a)\ []\ is$

$guard_e\ L\ pn\ ss\ is\ (UnOp\ uop\ e\ T) =$
$guard_e\ L\ pn\ ss\ is\ e \sqcap U_g\ (uop, typ\ e)\ (\lambda s_a.\ the\text{-}Prim\ (abs_e\ L\ pn\ ss\ []\ e\ s_a))$

$guard_e\ L\ pn\ ss\ is\ (BinOp\ bop\ e_1\ e_2\ T) =$
$guard_e\ L\ pn\ ss\ []\ e_1 \sqcap guard_e\ L\ pn\ ss\ []\ e_2 \sqcap$
$B_g\ (bop, typ\ e_1, typ\ e_2)\ (\lambda s_a.\ the\text{-}Prim\ (abs_e\ L\ pn\ ss\ []\ e_1\ s_a))\ (\lambda s_a.\ the\text{-}Prim\ (abs_e\ L\ pn\ ss\ []\ e_2\ s_a))$

$guard_e\ L\ pn\ ss\ is\ (LazyBinOp\ bop\ e_1\ e_2\ T) =$
$\mathsf{let}\ g_1 = guard_e\ L\ pn\ ss\ []\ e_1;$
$\quad g_2 = guard_e\ L\ pn\ ss\ []\ e_2$
$\mathsf{in\ case}\ bop\ \mathsf{of}\ logical\text{-}and \Rightarrow g_1 \sqcap (\lfloor\{s_a.\ abs_e\ L\ pn\ []\ []\ e_1\ s_a = Prim\ (Bool\ False)\}\rfloor \sqcup g_2)$
$\quad \mid logical\text{-}or \Rightarrow g_1 \sqcap (\lfloor\{s_a.\ abs_e\ L\ pn\ []\ []\ e_1\ s_a = Prim\ (Bool\ True)\}\rfloor \sqcup g_2)$

$guard_i :: ('s \Rightarrow nat\ list \Rightarrow val) \Rightarrow ('s \Rightarrow nat)\ list \Rightarrow ('s \Rightarrow nat)\ list \Rightarrow 's\ set\ option$
$guard_i\ arr\ rs\ [] \quad = None$
$guard_i\ arr\ rs\ (i{\cdot}is) = \lfloor\{s_a.\ i\ s_a < |the\text{-}Arrv\ (arr\ s_a\ (aps\ rs\ s_a))|\}\rfloor \sqcap guard_i\ arr\ (rs\ @\ [i])\ is$

Figure 8.3: Guarding expressions

Guard generation follows the same scheme as abs_e and augments ss and is as it comes along a structure or array access. The array bound checks are not introduced by the array access $ArrAcc\ e\ i\ T$ but when the array itself is reached. This is usually a variable or heap lookup, or rarely a literal value. The auxiliary function $guard_i$ then generates the indexing guards for all the accumulated indexes is.

For literal values and variable access the guard generation is handled by $guard_i$.

For array access *ArrAcc e i T* the abstracted index is augmented to the list *is* in order to guard expression *e*. Moreover, the index *i* is guarded, too.

For structure access *StructAcc e fn T* the field name *fn* is augmented to *ss* to guard *e*. For example, for an array of pairs and an expression `A[i].fst` this generates a guard for list *A-fst* in the corresponding Simpl state. That way the information about the affected sub-variable is propagated to *V* or *H*, respectively.

For dereferencing pointers with *Deref e T*, expression *e* is guarded and the resulting reference is tested against *NULL*.

For unary and binary operation the sub-expressions are guarded and the guard functions U_g and B_g are consulted.

For lazy binary operations we optimise this scheme, to allow a C coding stereotype. The test for a null pointer is often part of a Boolean expression that also dereferences the pointer:

```
if p != null && p->x < 5 ...
```

Since the second conjunct is only executed if the first conjunct holds this expression is safe. For the guard generation this means that we can regard the first part of the conjunction to be true while testing the guard for the second expression. This implication is encoded as union $\sqcup$ in $guard_e$, according to the tautology: $(P \longrightarrow Q) = (\neg\, P \vee Q)$.

The approach to propagate the index list through the term, may appear unnecessarily complicated. It may be more intuitive to omit the index accumulator *is* completely and simply generate the guard while the array access *ArrAcc e i T* is processed. However, since abs_e uses the index accumulator, $guard_e$ uses the index accumulator, too. Both recursions are "synchronous". The main soundness theorem for expression abstraction (cf. Theorem 8.7) considers abs_e and $gaurd_e$ simultaneously. If the guard holds, then the value of the abstracted expression has to be the same as in C0. The syncronicity of both recursions leads to a clear inductive argument.

The auxiliary function $guard_i$ generates the guards for all the accumulated array accesses. The first argument is the abstraction of the array, the second one is an internal accumulator that reverses the third argument *is*. Consider a two dimensional array and the access `A[i][j]`. Both accesses have to be guarded:

$$\{s_a.\ i\ s_a < |A\ s_a| \wedge j\ s_a < |(A\ s_a)_{[i\ s_a]}|\}.$$

The selection of the current dimension is accumulated in argument *rs* of $guard_i$. Note that argument *arr* does not directly yield $A\ s_a$ but *Arrv* (*map ValConstr* ($A\ s_a$)), where *ValConstr* is a place holder for a proper *val*-constructor depending on the type of the array. The *Arrv* is cancelled by the destructor *the-Arrv* inserted by $guard_i$. Moreover, since we only need the length of the list, we can get rid of mapping the value constructor on the list, by simplifying with the equation:

$$|map\ f\ xs| = |xs|.$$

We have defined abstraction of C0 expressions to Simpl expressions together with the corresponding guard generation. Now we draw our attention to the desired correctness property of this translation. We want to simulate C0 expression evaluation in Simpl, or to be more precise:

> If the guard in Simpl holds, then C0 expression evaluation does not cause a runtime fault, and the expression values in C0 and Simpl coincide.

In this formulation the guards can fail more often than runtime faults in C0 occur. This is intended, since it depends on the representation of unary and binary operations, if more guards than runtime faults are needed. For example, if we employ unbounded arithmetic in the program logic, then the guards also watch for over- and underflows. In C0 however, over- and underflows are silent. In the extreme case of an unsatisfiable guard the simulation property trivially holds. So this correctness theorem does not protect us against the trivial translation of all the guards to $\{\!|\mathit{False}|\!\}$. However, such a vacuous translation is immediately detected the first time one attempts to prove a Simpl program correct, since the Hoare logic enforces to prove that the guards always hold. This is not possible for unsatisfiable guards.

Before we formulate the simulation theorem, we specify the required properties of the locale parameters. To minimise those requirements, we only specify the behaviour of *LV*, *GV* and *H* for atomic values. Compound values can be constructed from the atomic components as Theorem 8.6 suggests. To specify the properties we relate C0-states with the corresponding Simpl states and compare the effects of C0 operations on the C0-state and abstracted operations on the Simpl state. We cannot only relate exactly one Simpl state to a C0 state. The Simpl state space makes no distinction between global and local variables. All the local variables of all procedures are visible. In a C0 program however, there is only one active procedure frame at each point in the execution. The local variables of other procedures are not visible in this state. Therefore only the content of the local variables in the active frame has to coincide with the Simpl state. All the values of the local variables of other procedures are irrelevant and thus can have an arbitrary value. To abstract the state we introduce another parameter to the locale:

$$\mathit{abs}_s :: \mathit{pname} \Rightarrow \mathit{state} \Rightarrow \mathit{'s\ set}$$

It takes the name of the current procedure and a C0-state and yields the set of all related Simpl states. As example consider a program with two procedures, `p` and `q`, with local variables `C` and `I`, respectively, and the standard mapping to a Simpl record. The following is a proper abstraction relation for the state:

$$\begin{array}{l}\mathit{abs}_s\ \mathit{pn}\ s = \\ \textsf{if}\ \mathit{pn} = \text{''}p\text{''}\ \textsf{then}\ \{s_a.\ \forall b.\ \mathit{lvars}\ s\ \text{''}C\text{''} = \lfloor \mathit{Prim}\ (\mathit{Bool}\ b)\rfloor \longrightarrow C\ s_a = b\} \\ \textsf{else if}\ \mathit{pn} = \text{''}q\text{''} \\ \quad\quad \textsf{then}\ \{s_a.\ \forall i.\ \mathit{lvars}\ s\ \text{''}i\text{''} = \lfloor \mathit{Prim}\ (\mathit{Intg}\ i)\rfloor \longrightarrow I\ s_a = i\}\ \textsf{else}\ \mathit{UNIV}\end{array}$$

In the context of procedure `p` only the content of Boolean `C` has to coincide, in the context of `q` only the content of integer `I`.

The locale *lookup* also introduces the type environment *TE* and the typing *GT* for global variables and *LT* for local variables.

Definition 8.20 ► **locale** *lookup* = *guard* +
fixes
state abstraction:
$abs_s :: pname \Rightarrow state \Rightarrow 's\ set$
type environment:
$TE :: tname \rightharpoonup ty$
typing for global variables:
$GT :: vname \rightharpoonup ty$
typing for local variables:
$LT :: pname \Rightarrow vname \rightharpoonup ty$
assumes

(1) *Every atomic component of a local C0 variable can be obtained via LV from an abstract state:*

$$[\![lvars\ s\ vn = \lfloor v \rfloor;\ LT\ pn\ vn = \lfloor T \rfloor;\ \vdash_v v :: T;\ sel_T\ (T, ss) = \lfloor sT \rfloor;\ atomic_T\ sT;\ idxfits\ (sel_v\ (v, ss), is);\ s_a \in abs_s\ pn\ s]\!] \Longrightarrow LV\ pn\ vn\ ss\ s_a\ is = idx_v\ (sel_v\ (v, ss), is)$$

(2) *Every atomic component of a global C0 variable can be obtained via GV from an abstract state:*

$$[\![gvars\ s\ vn = \lfloor v \rfloor;\ GT\ vn = \lfloor T \rfloor;\ \vdash_v v :: T;\ sel_T\ (T, ss) = \lfloor sT \rfloor;\ atomic_T\ sT;\ idxfits\ (sel_v\ (v, ss), is);\ s_a \in abs_s\ pn\ s]\!] \Longrightarrow GV\ vn\ ss\ s_a\ is = idx_v\ (sel_v\ (v, ss), is)$$

(3) *Every atomic component of a C0 heap location can be obtained via H from an abstract state:*

$$[\![heap\ s\ l = \lfloor v \rfloor;\ TE\ tn = \lfloor T \rfloor;\ \vdash_v v :: T;\ sel_T\ (T, ss) = \lfloor sT \rfloor;\ atomic_T\ sT;\ idxfits\ (sel_v\ (v, ss), is);\ s_a \in abs_s\ pn\ s]\!] \Longrightarrow H\ tn\ ss\ (Rep\text{-}loc\ l)\ s_a\ is = idx_v\ (sel_v\ (v, ss), is)$$

(4) *If guard B_g holds, then the binary operation does not cause a runtime fault and B yields the same value as apply-binop:*

$$[\![\vdash_v Prim\ (v_1\ s_a) :: T_1;\ \vdash_v Prim\ (v_2\ s_a) :: T_2;\ bounded\ (v_1\ s_a);\ bounded\ (v_2\ s_a);\ T_1 \ll bop \gg T_2 :: T]\!] \Longrightarrow \text{case}\ apply\text{-}binop\ (bop, v_1\ s_a, v_2\ s_a)\ \text{of}\ None \Rightarrow \neg\ s_a \in\in B_g\ (bop, T_1, T_2)\ v_1\ v_2 \mid \lfloor v \rfloor \Rightarrow s_a \in\in B_g\ (bop, T_1, T_2)\ v_1\ v_2 \longrightarrow B\ (bop, v_1\ s_a, v_2\ s_a) = v$$

(5) *If guard U_g holds, then the unary operation does not cause a runtime fault and U yields the same value as apply-unop:*

$$[\![\vdash_v Prim\ (v_1\ s_a) :: T_1;\ bounded\ (v_1\ s_a);\ \ll uop \gg T_1 :: T]\!] \Longrightarrow \text{case}\ apply\text{-}unop\ (uop, v_1\ s_a)\ \text{of}\ None \Rightarrow \neg\ s_a \in\in U_g\ (uop, T_1)\ v_1 \mid \lfloor v \rfloor \Rightarrow s_a \in\in U_g\ (uop, T_1)\ v_1 \longrightarrow U\ (uop, v_1\ s_a) = v$$

The premises of the requirements (1)–(3) restrict the selector path *ss* and the index list *is* to sensible values. The selector path has to select an atomic component: $sel_T\ (T, ss) = \lfloor sT \rfloor$ and *atomic sT*. The index list has to fit to the selected value: $idxfits\ (sel_v\ (v, ss), is)$.

Since the basic lookup functions *LV*, *GV* and *H* are only specified for atomic values, we can only prove the correctness of the abstraction of expressions abs_e for atomic values. Note that this already includes all kinds of unary- and (lazy-)binary operations since they are only concerned with primitive and thus atomic values.

The following list describes the naming conventions and common entities of the theorems about the simulation of C0 in Simpl:

Π: C0 program

TE: type environment, maps type names to types usually: $TE = tnenv\ \Pi$

GT: global typing, maps global variable names to types

$LT\ pn$: local typing, maps local variable names to types, parametrised by the procedure name pn

$L = dom\ (LT\ pn)$: The domain of the local typing defines the set of local variables of a procedure. Parameters and the result variable are included.

$GT\ {+}{+}\ LT\ pn$: typing for procedure pn, in case of a conflict global types are overridden with local types

HT: heap typing, maps locations to type names

s: C0 state

s_a: Simpl state

$dom\ (lvars\ s)$: the domain of the local variables defines the set of assigned local variables in the current state

There is a small subtlety concerning the index list is. For the guard generation $guard_e$ the indexes are functions from a Simpl state to a natural number, whereas for abs_e they are plain natural numbers. The auxiliary functions aps and Ks convert between both representations. Function aps takes a list of functions and a state and applies them all to the state. Function Ks takes a list and converts it to a list of constant functions.

◂ Definition 8.21

$$\begin{aligned}
&aps :: ('s \Rightarrow 'a)\ list \Rightarrow 's \Rightarrow 'a\ list\\
&aps\ []\ s \quad = []\\
&aps\ (i{\cdot}is)\ s = i\ s{\cdot}aps\ is\ s
\end{aligned}$$

◂ Definition 8.22

$$\begin{aligned}
&Ks :: 'a\ list \Rightarrow ('s \Rightarrow 'a)\ list\\
&Ks\ [] \quad = []\\
&Ks\ (i{\cdot}is) = (\lambda s.\ i){\cdot}Ks\ is
\end{aligned}$$

Now we prove the main simulation theorem for evaluation of expressions with an atomic type. If the guard holds, then the C0 expression does not cause a runtime fault and evaluates to the same value as the Simpl abstraction. The theorem is generalised for the purpose of the inductive proof. For the intended core theorem let $ss = []$ and $is = []$. Hence $sT = T$ and T has to be atomic. Moreover, $dimfits\ sT\ []$ trivially holds.

◂ Theorem 8.7 (in *lookup*) *Simulation of atomic expression evaluation*

For a conforming C0-state s: $TE\vdash s :: HT,LT\ pn\restriction_A,GT$, where $TE = tnenv\ \Pi$, and an abstract Simpl-state $s_a \in abs_s\ pn\ s$, given an expression e that is welltyped: $\Pi,GT\ {+}{+}\ LT\ pn,HT\vdash_e e\ \surd$, and also definitely assigned: $\mathcal{D}_e\ e\ L\ A$, with respect to $A \subseteq dom\ (lvars\ s)$ and $L = dom\ (LT\ pn)$, given a selector path ss: $sel_T\ (typ\ e, ss) = \lfloor sT \rfloor$, to an atomic type sT: $atomic_T\ sT$, then for every proper index list is: $dimfits\ sT\ is$, we have:

$$\begin{aligned}
&\textsf{case}\ eval\ L\ s\ e\ \textsf{of}\ None \Rightarrow \neg\ s_a \in\in guard_e\ L\ pn\ ss\ is\ e\\
&\mid \lfloor v \rfloor \Rightarrow s_a \in\in guard_e\ L\ pn\ ss\ is\ e \longrightarrow\\
&\qquad abs_e\ L\ pn\ ss\ (aps\ is\ s_a)\ e\ s_a = idx_v\ (sel_v\ (v, ss), aps\ is\ s_a) \wedge\\
&\qquad s_a \in\in guard_e\ L\ pn\ ss\ is\ (Lit\ v\ (typ\ e)).
\end{aligned}$$

Proof. By induction on expression *e*. We omit a detailed proof here, but rather elaborate on how the different assumptions and constraints fit together. From the requirements on the lookup functions *LV*, *GV* and *H* in locale *lookup* (cf. Definition 8.20(1), 8.20(2), and 8.20(3)) we only get assumptions for defined and welltyped variable and heap values. By the subject reduction Theorem 7.14 we know that evaluation always yields welltyped values, since the expression itself is welltyped. For global variables and the heap, definedness of variables and locations is ensured by the conformance assumption of the C0-state *s*. For local variables the definite assignment analysis ensures that we only access assigned and hence defined variables. This is why $A \subseteq dom\ (lvars\ s)$ is required. The subject reduction theorems like Theorem 7.14 do not need this assumption since they implicitly consider executions where only defined variables are accessed. Otherwise the evaluation yields *None*.

Every proof of a single induction case basically follows the case distinction of the conclusion. In case the C0 expression causes a runtime fault, a certain (sub-)expression caused it and therefore we know by the induction hypothesis that the guard for this sub-expression does not hold. Hence the guard for the whole expression fails, too. For the case *ArrAcc a i T* this argumentation does not directly work, since guarding the array access is postponed to the guard of *a*, by augmenting the index list *is* with the current index, say *n*. To be able to use the information from the postponed guard at the position of the array access the last conjunct in the conclusion of the theorem was introduced: The guard holds for the evaluated literal value, too. Now we can argue that the guard holds for the evaluated array value for all indexes $n{\cdot}is$, if it holds for the array access. Hence if an array bound violation occurs it is detected by this guard.

In case the C0 expression causes no runtime-fault and evaluates to $\lfloor v \rfloor$, we can assume that the guard for the expression holds since we have to show an implication. Hence we can conclude that the guards for the sub-expressions hold. From the induction hypothesis we get that sub-expression evaluation is correctly simulated by the abstraction and can lift these results to the whole expression. Case *Lit* is trivial. Cases *VarAcc* and *Deref* are the base cases of the induction and the places where the specifications for *GV*, *LV* and *H* come in. We can only use them if $idxfits\ (sel_v\ (v, ss), is)$. We know that $guard_i$ holds for all the accumulated indexes *is*. Moreover, from the assumptions we know that the dimension of the index lists fits. By induction on *is* we show that *idxfits* for all the *is* and thus the specifications can be employed to prove the simulation.

The simulation of unary and binary operations is handled by the Requirements 8.20(5) and 8.20(4). □

Theorem 8.7 allows to simulate evaluation of an expression of an atomic type. For assignments and procedure calls also compound types can occur. We can construct a compound value out of its atomic components as in Theorem 8.6. We also have to guard the expression evaluation to ensure that no runtime faults can occur. Intuitively it is sufficient to generate only one guard for a proper atomic component. For example, let `A` be an array of pairs. To guard an expression `A[i]`, it is sufficient to generate a guard for either array *A-fst* or *A-snd*. Both arrays have the same length, since they are split versions of an array of pairs. One guard protects both arrays.

The following lemma expresses that a guard stays valid if we switch the selector path to an atomic component.

◂ Lemma 8.8 (in *lookup*)

For a conforming C0-state s: $TE\vdash s :: HT,LT\ pn{\restriction}_A,GT$, where $TE = tnenv\ \Pi$, and an abstract Simpl-state $s_a \in abs_s\ pn\ s$, given an expression e that is welltyped: $\Pi,GT ++ LT\ pn,HT\vdash_e e\ \surd$, and also definitely assigned: $\mathcal{D}_e\ e\ L\ A$, with respect to $A \subseteq dom\ (lvars\ s)$ and $L = dom\ (LT\ pn)$, for any two selector paths ss and ss′ ($sel_T\ (typ\ e, ss) = \lfloor sT \rfloor$, $sel_T\ (typ\ e, ss') = \lfloor sT' \rfloor$) to atomic types sT and sT′ ($atomic_T\ sT$, $atomic_T\ sT'$), we have:

If $s_a \in\in guard_e\ L\ pn\ ss\ []\ e$ then also $s_a \in\in guard_e\ L\ pn\ ss'\ []\ e$.

Proof. The lemma has to be generalised to be proven by induction on *e*. The generalised version is Lemma 8.12. It is postponed since it uses some auxiliary notions that are introduced in the following section. □

With this lemma we can extend Theorem 8.7 from atomic to arbitrary types. One guard for an arbitrary selector path *ss* has to be provided. The compound value is constructed by first mapping abs_e to all the *selectors* of the type. This yields the exploded version of the value. By imploding it we obtain the value.

◂ Theorem 8.9 (in *lookup*) *Simulation of expression evaluation*

For a conforming C0-state s: $TE\vdash s :: HT,LT\ pn{\restriction}_A,GT$, where $TE = tnenv\ \Pi$, and an abstract Simpl-state $s_a \in abs_s\ pn\ s$, given an expression e that is welltyped: $\Pi,GT ++ LT\ pn,HT\vdash_e e\ \surd$, and also definitely assigned: $\mathcal{D}_e\ e\ L\ A$, with respect to $A \subseteq dom\ (lvars\ s)$ and $L = dom\ (LT\ pn)$, given a selector path ss: $sel_T\ (typ\ e, ss) = \lfloor sT \rfloor$, to an atomic type sT: $atomic_T\ sT$, then we have:

case $eval\ L\ s\ e$ of $None \Rightarrow \neg\ s_a \in\in guard_e\ L\ pn\ ss\ []\ e$
$|\ \lfloor v \rfloor \Rightarrow s_a \in\in guard_e\ L\ pn\ ss\ []\ e \longrightarrow$
$\quad implode\ (typ\ e, map\ (\lambda ss.\ abs_e\ L\ pn\ ss\ []\ e\ s_a)\ (selectors\ (typ\ e))) = v.$

Proof. From Lemma 8.2 we know that the *selectors* of a type all yield a path to an atomic type. Since the index list supplied to abs_e is empty, Theorem 8.7 guarantees that every abstraction yields the corresponding atomic C0 value component. All components together make up the exploded version of the value and can be imploded to the original value via Theorem 8.6. □

The last contribution to the simulation of expressions is to lift the results obtained so far to expression lists. To guard an expression list, we pick the guard for the head of the *selectors* of each type. Note that *selectors* always returns at least one selector path for every type. For atomic types it is the empty list.

◂ Definition 8.23 (in *guard*)

$guard_{es}$:: vname set ⇒ pname ⇒ ty expr list ⇒ ′s set option
$guard_{es}\ L\ pn\ [] = None$
$guard_{es}\ L\ pn\ (e{\cdot}es) = guard_e\ L\ pn\ (hd\ (selectors\ (typ\ e)))\ []\ e \sqcap guard_{es}\ L\ pn\ es$

◂ Theorem 8.10 (in *lookup*) *Simulation of expression list evaluation*

For a conforming C0-state s: $TE\vdash s :: HT,LT\ pn{\restriction}_A,GT$, where $TE = tnenv\ \Pi$, and an abstract Simpl-state $s_a \in abs_s\ pn\ s$, given an expression list es that is welltyped: $\Pi,GT ++ LT\ pn,HT[\vdash_e]\ es\ \surd$, and where all expressions are definitely assigned: $\forall e\in set\ es.\ \mathcal{D}_e\ e\ L\ A$, with respect to $A \subseteq dom\ (lvars\ s)$ and $L = dom\ (LT\ pn)$, then we have:

case $evals\ L\ s\ es$ of $None \Rightarrow \neg\ s_a \in\in guard_{es}\ L\ pn\ es$
$|\ \lfloor vs \rfloor \Rightarrow$
$\quad s_a \in\in guard_{es}\ L\ pn\ es \longrightarrow$
$\quad map\ (\lambda e.\ implode\ (typ\ e, map\ (\lambda ss.\ abs_e\ L\ pn\ ss\ []\ e\ s_a)\ (selectors\ (typ\ e))))\ es = vs.$

Proof. By induction on the expression list *es* and Theorem 8.9. □

8.3 Left-Expressions

In C0, *leval* evaluates left-expression to a reduced left-expression, where array indexes are evaluated to an integer and in case of dereferencing a pointer, its address is calculated. This reduced left-expression is then used to guide function *assign* to the position where the new value is inserted. In Simpl we follow the same two steps. First, we calculate a left-value that corresponds to the reduced left-expression and then we use it to perform the state update. Splitting the values in Simpl leads to a normalisation of left-values. First all the field-selections are applied, since they determine which heap or variable to address, and then the array indexes are applied. We introduce the type of (memory-)*cell*s and left-values *lval* to record this information. The memory cell determines the C0-location that is affected by an assignment. A cell holds a complete C0-value. The left-value then additionally records the path inside this value to a sub-component that is replaced by the assignment.

Definition 8.24 ► *Memory cell*

A memory cell is either

- *a variable Var pn vn, where pn* :: *pname and vn* :: *vname, or*
- *a heap location Heap tn r, where tn* :: *tname and r* :: *ref.*

Definition 8.25 ► *Left value*

A left-value lval is of the form LVal c ss is, where c :: *cell and ss* :: *fname list and is* :: *nat list.*

The function *lval* extracts the left-value from a reduced left-expression. While it traverses the left-expression, the accumulator parameters *ss* and *is* are augmented by every structure or array access. The leaf of a left-expression is either a variable access *VarAcc vn T* or a pointer dereference: *Deref e T*. These leaf positions determine the kind of memory *cell* the resulting left-value gets.

Definition 8.26 ►

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{lval} :: \mathit{pname} \Rightarrow \mathit{fname\ list} \Rightarrow \mathit{nat\ list} \Rightarrow \mathit{ty\ expr} \Rightarrow \mathit{lval}} \\
\mathit{lval\ pn\ ss\ is}\ (\mathit{VarAcc\ vn\ T}) & = \mathit{LVal}\ (\mathit{Var\ pn\ vn})\ \mathit{ss\ is} \\
\mathit{lval\ pn\ ss\ is}\ (\mathit{ArrAcc\ e\ i\ T}) & = \mathit{lval\ pn\ ss}\ (\mathit{the\text{-}Unsgnd}_v\ (\mathit{the\text{-}Lit\ i}) \cdot \mathit{is})\ e \\
\mathit{lval\ pn\ ss\ is}\ (\mathit{StructAcc\ e\ fn\ T}) & = \mathit{lval\ pn}\ (\mathit{fn} \cdot \mathit{ss})\ \mathit{is\ e} \\
\mathit{lval\ pn\ ss\ is}\ (\mathit{Deref\ e\ T}) & = \mathit{LVal}\ (\mathit{Heap}\ (\mathit{tname}\ (\mathit{typ\ e}))\ (\mathit{the\text{-}Ref}\ (\mathit{the\text{-}Lit\ e})))\ \mathit{ss\ is} \\
\mathit{lval\ pn\ ss\ is}\ \text{-} & = \mathit{LVal\ arbitrary\ ss\ is}
\end{array}
$$

Note that the initial *ss* and *is* of *lval pn ss is le* appear as a suffix in the resulting left-value. Left-expression *le* adds its inherent selectors and indexes to the front. So if *ss'* and *is'* stem from the left-expression *le*: *lval pn* [] [] *le* = *LVal c ss' is'*, then we have *lval pn ss is le* = *LVal c* (*ss'* @ *ss*) (*is'* @ *is*). The last equation in the definition of *lval* is an extension to non left-expressions that maintains this property.

The abstraction abs_l takes a left-expressions and yields a left-value *lval*. Again the selection path *ss* and the index list *is* are used as accumulators. Initially they can be considered to be empty. As the left-expression is traversed, the array indices and the address of a pointer is evaluated with abs_e.

◂ Definition 8.27 (in *expr*)

$$
\begin{array}{ll}
abs_l :: vname\ set \Rightarrow pname \Rightarrow fname\ list \Rightarrow nat\ list \Rightarrow ty\ expr \Rightarrow 's \Rightarrow lval & \\
abs_l\ L\ pn\ ss\ is\ (VarAcc\ vn\ T)\ s_a & = LVal\ (Var\ pn\ vn)\ ss\ is \\
abs_l\ L\ pn\ ss\ is\ (ArrAcc\ e\ i\ T)\ s_a & = \mathsf{let}\ n = the\text{-}Unsgnd_v\ (abs_e\ L\ pn\ []\ []\ i\ s_a) \\
 & \quad \mathsf{in}\ abs_l\ L\ pn\ ss\ (n{\cdot}is)\ e\ s_a \\
abs_l\ L\ pn\ ss\ is\ (StructAcc\ e\ fn\ T)\ s_a & = abs_l\ L\ pn\ (fn{\cdot}ss)\ is\ e\ s_a \\
abs_l\ L\ pn\ ss\ is\ (Deref\ e\ T)\ s_a & = \mathsf{let}\ r = the\text{-}Ref\ (abs_e\ L\ pn\ []\ []\ e\ s_a); \\
 & \qquad tn = tname\ (typ\ e) \\
 & \quad \mathsf{in}\ LVal\ (Heap\ tn\ r)\ ss\ is
\end{array}
$$

Abstraction abs_l only uses abs_e for expressions of atomic types, namely integers and pointers. The only potential source of runtime faults is the evaluation of the indexes and addresses. Array bound and null pointer tests are not performed during left-expression evaluation but later on by the assignment. For evaluation of indexes and addresses via abs_e, both the selector path and the index list are []. The parameters *ss* and *is* of abs_l do not contribute to these applications. They are passed down to the leaf of the left-expression, where they are built into the left-value. Hence a guard for left-expression *le* for any selector path *ss* and any index list *is* is sufficient to guard the nested abs_e applications. For the selector paths this is crucial, since it allows to guard all the atomic assignments of a compound value by only one guard. This is built into the following theorem, by using different selector paths for the guard and for abs_l.

◂ Theorem 8.11 (in *lookup*) *Simulation of left-expression evaluation*

For a conforming C0-state s: $TE \vdash s :: HT,LT\ pn{\restriction_A},GT$, where $TE = tnenv\ \Pi$, and an abstract Simpl-state $s_a \in abs_s\ pn\ s$, given a left-expression le that is welltyped: $\Pi,GT ++ LT\ pn,HT \vdash_l le\ \surd$, and also definitely assigned: $\mathcal{D}_l\ le\ L\ A$, with respect to $A \subseteq dom\ (lvars\ s)$ and $L = dom\ (LT\ pn)$, then for any selector paths ss and ss' and index list is we have:

$$
\begin{array}{l}
\mathsf{case}\ leval\ L\ s\ le\ \mathsf{of}\ None \Rightarrow \neg\ s_a \in\in guard_e\ L\ pn\ ss\ is\ le \\
\mid \lfloor lv \rfloor \Rightarrow \\
\quad s_a \in\in guard_e\ L\ pn\ ss\ is\ le \longrightarrow \\
\quad abs_l\ L\ pn\ ss'\ (aps\ is\ s_a)\ le\ s_a = lval\ pn\ ss'\ (aps\ is\ s_a)\ lv\ \wedge \\
\quad s_a \in\in guard_e\ L\ pn\ ss\ is\ lv.
\end{array}
$$

Proof. By induction on left-expression *le*. Welltypedness and definite assignment of left-expression *le* ensure that every nested array index and pointer that is dereferenced is welltyped and definitely assigned. Moreover, the guard ensures that those sub-expressions are also correctly guarded. Since integers and addresses are both atomic types Theorem 8.7 is applicable to derive the simulation. □

Some more remarks on the theorem. Even if we guard left-expression *le* with $guard_e\ L\ pn\ []\ []\ le$, this guard can be more restrictive as necessary, since it also checks for array bound violations at *ArrAcc* and for dereferencing a null-pointer at *Deref*. On the contrary, *leval* does not cause runtime faults in those cases. However, since the evaluation of the left-expression is only a prelude to an assignment, where those runtime faults have to be excluded, we do not need to introduce a more liberal guard-generation. In the end we have to introduce the fully fledged guard for the assignment anyway.

Moreover, the last line in the conclusion is not only a technical strengthening of the conjecture to get a sufficient induction hypothesis as in case of Theorem 8.7. It is used to link the guard for the left-expression to the following assignment. We know that the guard also holds for the resulting left-value and hence array bounds and null-pointers are sufficiently protected to ensure a smooth execution of the assignment according to this left-value.

Now we come back to Lemma 8.8. It allows to generate only one guard for an arbitrary atomic component of a type, since this guard also protects all the other components. This is also a crucial building block for assignments of compound values. Since the single assignment in C0 is split up into a series of assignments of all the atomic components in Simpl. The type of a left-expression that is obtained by *typ le* is the same as if we type *le* as ordinary (right-)expressions. It is the type of the cell-component where the new value is inserted. In case of an integer array `A`, the left expression `A[i]` has type integer. We introduce the notion of a *cell type* that yields the type of the (complete) cell, where the assignment takes place. In case of the example `A[i]`, the cell type is the type of the array `A`: "integer array". For non left-expressions the cell-type coincides with the type.

Definition 8.28 ►

$$\begin{array}{ll} \mathit{cty} :: \mathit{ty}\ \mathit{expr} \Rightarrow \mathit{ty} & \\ \mathit{cty}\ (\mathit{VarAcc}\ \mathit{vn}\ T) & = T \\ \mathit{cty}\ (\mathit{ArrAcc}\ e\ i\ T) & = \mathit{cty}\ e \\ \mathit{cty}\ (\mathit{StructAcc}\ e\ \mathit{cn}\ T) & = \mathit{cty}\ e \\ \mathit{cty}\ (\mathit{Deref}\ e\ T) & = T \\ \mathit{cty}\ e & = \mathit{expr\text{-}ty}\ e \end{array}$$

The following lemma is a generalisation of Lemma 8.8 for which the induction on expression *e* works out. Specialising *css* to [] yields Lemma 8.8 since the remaining assumptions get trivial. Lemma 8.8 states that if a guard for a selector path *ss* to an atomic type holds, then the guard for any other selector path *ss'* to an atomic type also holds. To make the induction work for structure access, where the selector path is extended, we generalise the selector paths to express that we are "somewhere in-between" the cell type *cty e* and the type of the atomic component. Selector path *ess* is inherent to expression *e*. It bridges from cell type *cty e* to expression type *typ e*. Moreover, there is a common selector path *css* that selects a component type beginning at expression type *typ e*. It can be extended to an atomic type by *ss* and *ss'*. Function *lval* is only used to get hold of *ess* and *eis*, the selectors and the indexes that are inherent to expression *e*. It imposes no constraints.

Lemma 8.12 ► (in *lookup*)

For a conforming C0-state s: $TE \vdash s :: HT, LT\ pn{\upharpoonright_A}, GT$, *where* $TE = \mathit{tnenv}\ \Pi$, *and an abstract Simpl-state* $s_a \in \mathit{abs}_s\ pn\ s$, *given an expression e that is welltyped:* $\Pi, GT \mathbin{+\!+} LT\ pn, HT \vdash_e e\ \surd$, *and also definitely assigned:* $\mathcal{D}_e\ e\ L\ A$, *with respect to an* $A \subseteq \mathit{dom}\ (\mathit{lvars}\ s)$ *and* $L = \mathit{dom}\ (LT\ pn)$, *given a common selector path css:* $\mathit{sel}_T\ (\mathit{typ}\ e, \mathit{css}) = \lfloor \mathit{csT} \rfloor$ *that can be extended by two selector paths ss and ss'* ($\mathit{sel}_T\ (\mathit{csT}, \mathit{ss}) = \lfloor sT \rfloor$, $\mathit{sel}_T\ (\mathit{csT}, \mathit{ss}') = \lfloor sT' \rfloor$) *to atomic types sT and sT'* ($\mathit{atomic}_T\ sT$, $\mathit{atomic}_T\ sT'$), *let* $\mathit{lval}\ pn\ []\ []\ e = \mathit{LVal}\ c\ \mathit{ess}\ \mathit{eis}$, $\mathit{sel}_T\ (\mathit{cty}\ e, \mathit{ess} @ \mathit{css}) = \lfloor \mathit{esT} \rfloor$ *and let is be a proper index list:* $\mathit{idx}_T\ (\mathit{esT}, \mathit{Ks}\ \mathit{eis} @ \mathit{is}) = \lfloor iT \rfloor$, *then we have:*

If $s_a \in\in \mathit{guard}_e\ L\ pn\ (\mathit{css} @ \mathit{ss})\ \mathit{is}\ e$ *then also* $s_a \in\in \mathit{guard}_e\ L\ pn\ (\mathit{css} @ \mathit{ss}')\ \mathit{is}\ e$.

Proof. By induction on expression *e*. For the base cases *VarAcc*, *Deref* and *Lit* the corresponding conjecture for guard_i is proven by induction on the accumulated index list *is*. □

8.4 Assignments

C0 assignments *Ass le e* in Simpl are performed in three steps:

- Calculate the left-value of *le* with abs_l.
- Calculate all atomic values of *e* with abs_e.
- Update the state by a sequence of updates with the atomic values.

A single state update in C0 is split to a sequence of atomic state updates in Simpl. We describe this sequence of state updates in C0 and prove that it simulates the original C0 state update. Then it is sufficient to require that the state updates of atomic components are simulated correctly.

The first auxiliary notion is $upd_v\ (v, ss, is, v')$. It updates value v at the position determined by selection path *ss* and index list *is* with the value v'.

◂ Definition 8.29

$$
\begin{array}{lcl}
upd_v :: val \times fname\ list \times nat\ list \times val \Rightarrow val & & \\
upd_v\ (v, [], [], v') & = & v' \\
upd_v\ (Structv\ fvs, s{\cdot}ss, is, v') & = & Structv \\
 & & \quad (assoc\text{-}update\ fvs\ s \\
 & & \quad\ \ (upd_v\ (the\ (map\text{-}of\ fvs\ s), ss, is, v'))) \\
upd_v\ (Arrv\ vs, ss, i{\cdot}is, v') & = & Arrv\ (vs[i := upd_v\ (vs_{[i]}, ss, is, v')]) \\
upd_v\ (Arrv\ vs, s{\cdot}ss, [], Arrv\ vs') & = & Arrv\ (map\ (\lambda(v, v').\ upd_v\ (v, s{\cdot}ss, [], v'))\ (zip\ vs\ vs'))
\end{array}
$$

If both *ss* and *is* are [], then the update takes place at the root of the value. This means that v is replaced by v'. This equation marks the end of the recursion, where all selections and indexes have been processed and the final position for the update is reached. For a structure value *Structv fvs* and a non-empty selection path $s{\cdot}ss$, field s of the field-list *fvs* is updated by function *assoc-update*. For an array value *Arrv vs* and a non-empty index list $i{\cdot}is$ the value list *vs* is updated at position i. These first three equations are the canonical equations for C0-values. The last equation stems from splitting values, since selection distributes over indexing. The first array *Arrv vs* is an array of structured values, where we attempt to update a certain component of all array elements. Array *Arrv vs′* is an array of the new component values. The non-empty selector path $s{\cdot}ss$ determines the sub-components. All array elements *vs* are updated according to the component values in *vs′*.

Value update upd_v is compatible with sel_v and idx_v:

◂ Lemma 8.13

If $\vdash_v v :: T$ *and* $sel_T\ (T, ss) = \lfloor sT \rfloor$ *and* $idxfits\ (sel_v\ (v, ss), is)$ *and* $idx_T\ (sT, is) = \lfloor iT \rfloor$ *and* $\vdash_v v' :: iT$ *then* $idx_v\ (sel_v\ (upd_v\ (v, ss, is, v'), ss), is) = v'$.

Proof. By induction on the recursion-scheme of sel_T. □

Updating an atomic component of a memory cell can be described as taking the current value v_c of the memory cell and then updating it via $upd_v\ (v_c, ss, is, v)$. This is the basic intuition for the abstract update functions GV_u, LV_u and H_u for global- and local variables and the heap that we introduce in locale *assign*. The arguments of these functions are similar to the corresponding lookup functions. Additionally the new atomic value is passed. The functions take the current state and yield the updated state.

Definition 8.30 ► **locale** *assign* =
fixes
global variable update:
GV_u :: *vname* ⇒ *fname list* ⇒ *nat list* ⇒ *val* ⇒ *'s* ⇒ *'s*
local variable update:
LV_u :: *pname* ⇒ *vname* ⇒ *fname list* ⇒ *nat list* ⇒ *val* ⇒ *'s* ⇒ *'s*
heap update:
H_u :: *tname* ⇒ *fname list* ⇒ *ref* ⇒ *nat list* ⇒ *val* ⇒ *'s* ⇒ *'s*

defines
variable update:
V_u :: *vname set* ⇒ *pname* ⇒ *vname* ⇒ *fname list* ⇒*nat list*⇒ *val*⇒ *'s* ⇒ *'s*
V_u *L pn vn ss is v* ≡ if *vn* ∈ *L* then LV_u *pn vn ss is v* else GV_u *vn ss is v*

Like lookup *V* the update V_u decides whether a global or a local variable is concerned. The function abs_u takes a left-value and delegates the update to V_u or H_u, depending on the kind of memory cell of the left-value.

Definition 8.31 ► (in *assign*)

$$\begin{aligned} abs_u &:: vname\ set \Rightarrow lval \Rightarrow val \Rightarrow {'s} \Rightarrow {'s} \\ abs_u\ L\ (LVal\ (Var\ pn\ vn)\ ss\ is)\ v\ s_a &= V_u\ L\ pn\ vn\ ss\ is\ v\ s_a \\ abs_u\ L\ (LVal\ (Heap\ tn\ r)\ ss\ is)\ v\ s_a &= H_u\ tn\ ss\ r\ is\ v\ s_a \end{aligned}$$

With abs_u we can simulate an assignment to an atomic component. To simulate arbitrary assignments we sequence the atomic assignments of the components. The components are obtained by the *selectors* function. To specify the update functions GV_u, LV_u and H_u we describe their effect on a C0-state. For example, consider an update of local variable *vn*, where *lvars s vn* = ⌊*v*⌋. Let *ss* and *is* determine the atomic component of *v* that is updated with a value *v'*. Then the modification of the local variables is described by:

$$lvars\ s(vn \mapsto upd_v\ (v, ss, is, v'))$$

Note that according to operator precedence, the map update takes place in (*lvars s*) not in *s*. The update upd_v (*v, ss, is, v'*) works fine as long as the type of *v* fits to *ss* and *is*, and the type of *v'* fits to the selected component. However, if *v* is for example a Boolean value and *ss* = [*"fst"*], the whole specification makes no sense. In C0 these type constraints are ensured by static typing, and the subject reduction theorems transfer them to program execution. There is one issue about the abstraction of C0 to Simpl regarding parameter passing and initialisation of local variables. In Simpl local variables are not reset to *None* when a procedure is entered. They just keep the current value. Since the definite assignment analysis of C0 ensures that every local variable is assigned to before it is read, this translation is sound. To specify a procedure call in terms of C0 we just keep the local variable setting of the caller, when we enter a procedure. If the caller has a local variable *x* of type Boolean, and also the callee has a local variable *x*, but of a pair type, we arrive at the undesired situation described above. The value of the Boolean *x* is still in the store and we cannot properly describe what an update of component *"fst"* or *"snd"* means. Hence we cannot properly describe component-wise initialisation. The same issue occurs for parameter passing, since their names are also local and hence can change the type during a procedure call. However, for parameter passing as well as for the initialisation of a local variable, we know that the complete variable gets a value, not only parts of it. This is inherent to parameter passing and is guaranteed by the definite assignment analysis for local variables. We can make use of this property to remedy the situation. Additionally to the correctness of updating of atomic components, we require from *LV* that it can properly initialise variables. In case the left-value indicates an initialisation of a local variable we make use of this requirement for *LV*. Instead of splitting the assignment to a sequence of atomic updates we initialise it in one step.

Predicate *linit L lv* tests whether a left-value *lv* denotes an initialisation of a local variable. The variable name has to be among the local variables *L* and the selector path and the index list have to be empty.

◂ Definition 8.32

$$\begin{array}{lcl} \mathit{linit} :: \mathit{vname\ set} \Rightarrow \mathit{lval} \Rightarrow \mathit{bool} & & \\ \mathit{linit}\ L\ (\mathit{LVal}\ (\mathit{Var}\ \mathit{pn}\ \mathit{vn})\ []\ []) & = & \mathit{vn} \in L \\ \mathit{linit}\ L\ (\mathit{LVal}\ \text{-}\ \text{-}\ \text{-}) & = & \mathit{False} \end{array}$$

Function *ass T L lv v* s_a decides whether *lv* is a local variable initialisation or not. The value *v* is provided in an exploded version: as a function from selector path to atomic value. In case of a local variable initialisation the value is imploded and *LV* takes care of the initialisation. Otherwise a sequence of updates of the atomic components is applied via abs_u. The library function *foldl* describes an iteration over a list. The type argument *T* denotes the type of *v*. The selectors of type *T* are used to extend the selectors of the left-value *lv*, such that altogether an atomic component of the cell is addressed.

◂ Definition 8.33 (in *assign*)

$$\begin{array}{l} \mathit{ass} :: \mathit{ty} \Rightarrow \mathit{vname\ set} \Rightarrow \mathit{lval} \Rightarrow\ (\mathit{fname\ list} \Rightarrow \mathit{val}) \Rightarrow {}'s \Rightarrow {}'s \\ \mathit{ass}\ T\ L\ (\mathit{LVal}\ c\ \mathit{ss}\ \mathit{is})\ v\ s_a = \\ \textsf{if}\ \mathit{linit}\ L\ (\mathit{LVal}\ c\ \mathit{ss}\ \mathit{is})\ \textsf{then}\ \mathit{abs}_u\ L\ (\mathit{LVal}\ c\ \mathit{ss}\ \mathit{is})\ (\mathit{implode}\ (T,\ \mathit{map}\ v\ (\mathit{selectors}\ T)))\ s_a \\ \textsf{else}\ \mathit{foldl}\ (\lambda s_a\ \mathit{ss}'.\ \mathit{abs}_u\ L\ (\mathit{LVal}\ c\ (\mathit{ss}\ @\ \mathit{ss}')\ \mathit{is})\ (v\ \mathit{ss}')\ s_a)\ s_a\ (\mathit{selectors}\ T) \end{array}$$

The requirements on GV_u, LV_u and H_u are collected in locale *update*. They are specified as commutation properties for the C0 update, the corresponding Simpl update and the state abstraction abs_s. For example, for the update of an atomic component of a local variable we have the following commuting diagram:

$$\begin{tikzcd} s_a \arrow[r, "LV_u\ pn\ vn\ ss\ is\ v'\ s_a"] & s_a{}' \\ s \arrow[u, "abs_s\ pn"] \arrow[r, "s(\!|lvars := lvars\ s(vn \mapsto upd_v\ (v,\ ss,\ is,\ v'))|\!)"'] & s' \arrow[u, "abs_s\ pn"'] \end{tikzcd}$$

We can either perform the C0 operation from state *s* to *s′* and then abstract the resulting state to $s_a{}'$, or first abstract state *s* to s_a and then perform the abstract operation *LV*. Since abstraction of the state abs_s yields a set of corresponding Simpl states the commutation properties are defined on the level of sets, where the infix ′ is the set image operation:

$$f\ `\ A = \{f\ a.\ a \in A\}.$$

The value v_c is the cell-value, i.e. The current value of the memory cell that is affected by the update.

Definition 8.34 ▶ **locale** *update* = *lookup* + *assign* +
assumes

(1) *Update of an atomic component of a local variable commutes:*

$[\![lvars\ s\ vn = \lfloor v_c \rfloor;\ LT\ pn\ vn = \lfloor T \rfloor;\ \vdash_v v_c :: T;\ sel_T\ (T, ss) = \lfloor sT \rfloor;$
$atomic_T\ sT;\ idx_T\ (sT, is) = \lfloor iT \rfloor;\ idxfits\ (sel_v\ (v_c, ss), is);\ \vdash_v v :: iT]\!]$
$\Longrightarrow LV_u\ pn\ vn\ ss\ is\ v\ `\ abs_s\ pn\ s$
$\subseteq abs_s\ pn\ (s(\!|lvars := lvars\ s(vn \mapsto upd_v\ (v_c, ss, is, v))|\!))$

(2) *Initialisation of a local variable commutes:*

$[\![LT\ pn\ vn = \lfloor T \rfloor;\ \vdash_v v :: T]\!]$
$\Longrightarrow LV_u\ pn\ vn\ []\ []\ v\ `\ abs_s\ pn\ s \subseteq abs_s\ pn\ (s(\!|lvars := lvars\ s(vn \mapsto v)|\!))$

(3) *Update of an atomic component of a global variable commutes:*

$[\![gvars\ s\ vn = \lfloor v_c \rfloor;\ GT\ vn = \lfloor T \rfloor;\ \vdash_v v_c :: T;\ sel_T\ (T, ss) = \lfloor sT \rfloor;\ atomic_T\ sT;$
$idx_T\ (sT, is) = \lfloor iT \rfloor;\ idxfits\ (sel_v\ (v_c, ss), is);\ \vdash_v v :: iT]\!]$
$\Longrightarrow GV_u\ vn\ ss\ is\ v\ `\ abs_s\ pn\ s$
$\subseteq abs_s\ pn\ (s(\!|gvars := gvars\ s(vn \mapsto upd_v\ (v_c, ss, is, v))|\!))$

(4) *Update of an atomic component of a heap location commutes:*

$[\![heap\ s\ l = \lfloor v_c \rfloor;\ TE\ tn = \lfloor T \rfloor;\ \vdash_v v_c :: T;\ sel_T\ (T, ss) = \lfloor sT \rfloor;\ atomic_T\ sT;$
$idx_T\ (sT, is) = \lfloor iT \rfloor;\ idxfits\ (sel_v\ (v_c, ss), is);\ \vdash_v v :: iT]\!]$
$\Longrightarrow H_u\ tn\ ss\ (Rep\text{-}loc\ l)\ is\ v\ `\ abs_s\ pn\ s$
$\subseteq abs_s\ pn\ (s(\!|heap := heap\ s(l \mapsto upd_v\ (v_c, ss, is, v))|\!))$

As in locale *lookup* the premises of the commutation properties restrict the selector path *ss* and *is* to sensible values. Moreover, the new value *v* has to fit to type *iT* of the selected sub-component. Apart from initialisation of local variables, the commutation requirements in locale *update* all assume that the cell already stores a proper value for which we attempt to update a component. In C0 this is ensured by conformance of the state and the definite assignment analysis. To talk uniformly about local, global and heap values and updates we introduce some auxiliary functions on the level of cells. First, with val_c we obtain the value that is stored in the memory cell, in case it is defined and not the cell corresponding to the null pointer.

Definition 8.35 ▶

$val_c :: vname\ set \Rightarrow cell \Rightarrow state \Rightarrow val\ option$
$val_c\ L\ c\ s \equiv$
$\textsf{case}\ c\ \textsf{of}\ Var\ pn\ vn \Rightarrow lookup\text{-}var\ L\ s\ vn$
$|\ Heap\ tn\ r \Rightarrow \textsf{if}\ r = NULL\ \textsf{then}\ None\ \textsf{else}\ heap\ s\ (Abs\text{-}loc\ r)$

Predicate $null_c$ tests whether a cell corresponds to the null pointer.

Definition 8.36 ▶

$null_c :: cell \Rightarrow bool$
$null_c\ c \equiv \textsf{case}\ c\ \textsf{of}\ Var\ pn\ vn \Rightarrow False\ |\ Heap\ tn\ r \Rightarrow r = NULL$

In some places it is more convenient to get rid of the option layer in val_c. We introduce the function *dval T v* that returns the default value of type *T* in case *v* = *None*.

Definition 8.37 ▶

$dval :: ty \Rightarrow val\ option \Rightarrow val$
$dval\ T\ v \equiv \textsf{case}\ v\ \textsf{of}\ None \Rightarrow default\text{-}val\ T\ |\ \lfloor v \rfloor \Rightarrow v$

The function $dval_c$ returns a default value of the cell in case the value is not defined. In contrast to *the* ($val_c\ L\ c\ s$), $dval_c\ T\ L\ c\ s$ still yields a welltyped value of type *T* in case of an undefined cell.

$dval_c :: ty \Rightarrow vname\ set \Rightarrow cell \Rightarrow state \Rightarrow val$
$dval_c\ T\ L\ c\ s \equiv dval\ T\ (val_c\ L\ c\ s)$ ◂ Definition 8.38

With $upd_c\ L\ c\ v\ s$ an update of a cell c in state s with the new value v is performed.

$upd_c :: vname\ set \Rightarrow cell \Rightarrow val \Rightarrow state \Rightarrow state$ ◂ Definition 8.39
$upd_c\ L\ c\ v\ s \equiv$
case c of $Var\ pn\ vn \Rightarrow update\text{-}var\ L\ s\ vn\ v$
$\mid Heap\ tn\ r \Rightarrow s(\!|heap := heap\ s(Abs\text{-}loc\ r \mapsto v)|\!)$

From these definitions it is obvious that for non-null cells, the value stored by upd_c can be extracted by val_c.

If $\neg\ null_c\ c$ *then* $val_c\ L\ c\ (upd_c\ L\ c\ v\ s) = \lfloor v \rfloor$. ◂ Lemma 8.14

If $\neg\ null_c\ c$ *then* $dval_c\ T\ L\ c\ (upd_c\ L\ c\ v\ s) = v$. ◂ Lemma 8.15

Finally function *upd* defines the update on the level of left-values. It takes the current value of the cell, updates a component of it with function upd_v according to the selection path and the index list of the left-value and finally stores the new value in the memory cell.

$upd :: ty \Rightarrow vname\ set \Rightarrow lval \Rightarrow val \Rightarrow state \Rightarrow state$ ◂ Definition 8.40
$upd\ T\ L\ (LVal\ c\ ss\ is)\ v\ s = upd_c\ L\ c\ (upd_v\ (dval_c\ T\ L\ c\ s, ss, is, v))\ s$

The function *upd* provides the first basic bridge between assignments in C0 and the corresponding Simpl version. Firstly we can immediately generalise the requirements in locale *update* to the level of cells:

Let $val_c\ L\ c\ s = \lfloor v_c \rfloor$ *and* $L = dom\ (LT\ pn)$. *Given the following typing of cell* c*:* ◂ Lemma 8.16 (in *update*)

case c of $Var\ pn'\ vn \Rightarrow pn' = pn \wedge (GT\ {+}{+}\ LT\ pn)\ vn = \lfloor T \rfloor$
$\mid Heap\ tn\ r \Rightarrow TE\ tn = \lfloor T \rfloor$

Let $\vdash_v v_c :: T$. *Given a selector path* ss*:* $sel_T\ (T, ss) = \lfloor sT \rfloor$*, to an atomic type* sT*:* $atomic_T\ sT$*, and a proper index list* is *(*$idx_T\ (sT, is) = \lfloor iT \rfloor$*,* $idxfits\ (sel_v\ (v_c, ss), is)$*) and a welltyped component value* v*:* $\vdash_v v :: iT$*, then:*
$abs_u\ L\ (LVal\ c\ ss\ is)\ v\ `\ abs_s\ pn\ s \subseteq abs_s\ pn\ (upd\ T\ L\ (LVal\ c\ ss\ is)\ v\ s)$.

And secondly we can characterise the effect of *assign* with *upd*.

For a conforming C0-state s*:* $TE \vdash s :: HT,LT\ pn \restriction_A,GT$*, where* $TE = tnenv\ \Pi$*, given a welltyped left expression* le*:* $\Pi,GT\ {+}{+}\ LT\ pn,HT \vdash_e le\ \surd$ *that is definitely assigned:* $\mathcal{D}_l\ le\ L\ A$ *with respect to* $L = dom\ (LT\ pn)$ *and have* reduced le*, then we have:* ◂ Lemma 8.17
If $assign\ L\ s\ le\ v = \lfloor s' \rfloor$ *then* $upd\ (cty\ le)\ L\ (lval\ pn\ []\ []\ le)\ v\ s = s'$.

Proof. We first generalise the conclusion to get a proper induction hypothesis for an intermediate position in the left-expression:

Let $v_d = dval\ (typ\ le)\ (eval\ L\ s\ le)$. If $assign\ (dom\ (LT\ pn))\ s\ le\ (upd_v\ (v_d, ss, is, v)) = \lfloor s' \rfloor$ then $upd\ (cty\ le)\ (dom\ (LT\ pn))\ (lval\ pn\ ss\ is\ le)\ v\ s = s'$.

With v_d we describe the value of the cell-component that is addressed by the left-expression *le*. The accumulated *ss* and *is* update a sub-component of this value via function upd_v. The *assign* updates the cell with this new cell-component. This situation is general enough for an induction on left-expression *le*. The proof is straightforward.

By instantiating ss = [] and is = [] in the generalised conjecture, the term upd_v (v_d, ss, is, v') simplifies to v and hence v_d itself does not appear anymore. We arrive at the original conjecture. □

The conclusions of Lemmas 8.16 and 8.17 show that function *upd* gives us a common base to compare the effect of *assign* in C0 and abs_u in Simpl. In Simpl the update is described from the view of the affected cell, whereas in C0 this cell is somehow disguised by the left-expression in *assign*. With *upd* we can bring the affected cell to light. Consider a reduced left-expression *le* to an atomic component, where *lval pn* [] [] *le* = *LVal c ss is* and *T* = *cty le*. This unifies the updates in Lemmas 8.16 and 8.17. Hence we can simulate an assignment to an atomic component in Simpl.

The following lemma connects the cell value, and the value (not the left-value) of a reduced left-expression *le*. Let *lval pn* [] [] *le* = *LVal c ss is*. If the left-expression evaluates without runtime-faults: *eval L le s* = $\lfloor v \rfloor$, then we can obtain a proper cell value val_c L c s = $\lfloor v_c \rfloor$. The value v is a sub-component of v_c, and we can select it via *ss* and *is*: idx_v (sel_v (v_c, ss), is) = v. This intuition was already used in the proof of Lemma 8.17. Since evaluation checks for dereferencing null pointers and array bounds, we also know that none of them are violated. Hence in case of a heap cell, the address is valid. This is already part of the definition of val_c. The absence of array bound violations results in *idxfits* (sel_v (v_c, ss), is). Moreover we obtain welltypedness of the cell-value, and the cell itself. It might be irritating that we care about value-evaluation of left-expressions at all, when thinking of assignments. The reason is the view on assignments in Simpl that can, for example, be seen in Lemma 8.16 or the definition of *upd*. We start from the cell-value, and first select the sub-value according to *ss* and *is*, where the update finally takes place. This selection corresponds to evaluation of the left-expression in C0. The results from the following lemma, like welltypedness of the cell-value and the cell, or that the indexes fit to the cell-value, are exactly the premises that we need for Lemma 8.16.

Lemma 8.18 ► *For a conforming C0-state s: TE⊢ s :: HT,LT pn*$\restriction_A$*,GT, where TE = tnenv Π, given a welltyped left expression le: Π,GT ++ LT pn,HT*$\vdash_e$ *le* $\surd$ *that is reduced: reduced le, and definitely assigned:* $\mathcal{D}_e$ *le L A with respect to L = dom (LT pn), moreover let lval pn* [] [] *le = LVal c ss is, then we have:*

If eval L s le = $\lfloor v \rfloor$ *then*
$\exists v_c.\ val_c\ L\ c\ s = \lfloor v_c \rfloor\ \wedge$
  $idx_v\ (sel_v\ (v_c, ss), is) = v\ \wedge$
  $idxfits\ (sel_v\ (v_c, ss), is)\ \wedge$
  $\lfloor HT \rfloor \vdash_v v_c :: cty\ le\ \wedge$
  case c of Var pn′ vn ⇒ pn′ = pn ∧ (GT ++ LT pn) vn = $\lfloor cty\ le \rfloor$
  | Heap tn r ⇒ HT (Abs-loc r) = $\lfloor tn \rfloor$ *∧ TE tn =* $\lfloor cty\ le \rfloor$*.*

Proof. By induction on *le*. Since *eval* does not cause a runtime fault we know that no array bound is violated by *le* and that the cell value must be defined. The third and fourth parts of the conjunction are used to handle the case of array- and structure access, respectively. The *idxfits* ensures that indexing with idx_v works correctly. The type of the cell-value is exploited to argue that the field name of a structure access is valid, so that sel_v yields a proper result. □

If evaluation of a reduced left-expression does not cause a runtime fault, then the corresponding assignment succeeds, too.

◄ Lemma 8.19

If reduced le and *eval L s le ≠ None* then *∀v. assign L s le v ≠ None.*

We can close the chain of argumentation, if we know that evaluation of the left-expression does not cause a runtime-fault. Then with Lemma 8.19 we also know that the assignment succeeds, and thus Lemma 8.17 is applicable. Moreover, since the evaluation is successful the preconditions of Lemma 8.16 can be discharged with Lemma 8.18.

From the simulation Theorem 8.7 for expressions we already know that evaluation does not cause a runtime fault if the corresponding guard holds. Hence we can already come up with a simulation lemma for assignments to atomic components.

◄ Lemma 8.20 (in *update*) *Simulation of atomic assignment*

For a conforming C0-state *s*: $TE\vdash s :: HT,LT\ pn\upharpoonright_A,GT$, where $TE = tnenv\ \Pi$ and an abstract Simpl-state: $s_a \in abs_s\ pn\ s$, given a welltyped and reduced left-expression *le* ($\Pi,GT\ {+}{+}\ LT\ pn,HT\vdash_e le\ \surd$ and *reduced le*) that is definitely assigned: $\mathcal{D}_e\ le\ L\ A$, with respect to $A \subseteq dom\ (lvars\ s)$ and $L = dom\ (LT\ pn)$, let $atomic_T\ (typ\ le)$ and $\vdash_v v :: typ\ le$, then we have:

case *assign L s le v* of *None* $\Rightarrow \neg\ s_a \in\in guard_e\ L\ pn\ []\ []\ le$
$\mid \lfloor s'\rfloor \Rightarrow s_a \in\in guard_e\ L\ pn\ []\ []\ le \longrightarrow abs_u\ L\ (lval\ pn\ []\ []\ le)\ v\ s_a \in abs_s\ pn\ s'$.

Proof. In case the assignment causes a runtime fault, the guard also fails, since otherwise Lemmas 8.7 and 8.19 lead to a contradiction. In case the assignment succeeds and the guard holds, we can discharge the preconditions of Lemma 8.16, since the evaluation of the left-expression succeeds according to 8.7 and hence Lemma 8.18 is applicable. The remaining gap to *assign* is closed by Lemma 8.17. □

It might be puzzling that the previous lemma uses the typing judgement and definite assignment analysis for ordinary expressions, instead of left-expressions. Generally the typing judgement for left-expressions is more restrictive then the one for expressions, since it rules out all the non left-expressions. However, *reduced le* already rules them out. In this case both typing judgements are equivalent:

◄ Lemma 8.21

If reduced le then $(\Pi,VT,HT\vdash_e le\ \surd) = (\Pi,VT,HT\vdash_l le\ \surd)$.

Proof. By induction on left-expression *le*. □

For the definite assignment analysis the situation is different. Here the analysis for expressions is strictly more restrictive as the one for left-expressions. In case of a variable access it guarantees that the variable is defined in the current state. Only in those defined cases the updated of a component is properly described by function upd_v applied to the cell-value and hence by function *upd*. The only situation were we have to deal with undefined memory cells is for initialisation of local variables and parameter passing. Keep in mind that they are handled in a different manner by function *ass* (cf. Definition 8.33 on p. 173). The initialisation is performed in a single step and not component-wise. The correctness of this initialisation is already required in the assumptions on LV_u in locale *update*. Otherwise definite assignment $\mathcal{D}_l$ and $\mathcal{D}_e$ coincides:

Lemma 8.22 ► *If reduced le and* $\neg$ *linit L (lval pn [] [] le) then* $\mathcal{D}_l$ *le L A* = $\mathcal{D}_e$ *le L A*.

Proof. By induction on left-expression *le*. □

The remaining step for the simulation of a general assignment is to lift Lemma 8.20 to non-atomic assignments. We argue that a non-atomic assignment can be simulated by a sequence of atomic assignments to the components.

We start with a basic lemma for upd_v. Consider two values v and v' of type T. If we start with v and iterate $upd_v\ (v, ss, [], sel_v\ (v', ss))$ over all selectors of T, then finally v is replaced by v'.

Lemma 8.23 ► *If* $\vdash_v v :: T$ *and* $\vdash_v v' :: T$ *and distinct-selectors T then*
$foldl\ (\lambda v\ ss.\ upd_v\ (v, ss, [], sel_v\ (v', ss)))\ v\ (selectors\ T) = v'$.

Proof. By induction on the recursion-scheme of *selectors T*. □

This lemma allows to replace the component of the cell-value that is updated, with a sequence of updates. Consider the assignment statement *Ass le e* and let *lval pn* [] [] *le* = *LVal c ss is* and $val_c\ L\ c\ s = \lfloor v_c \rfloor$ and let the value of *e* be v'. The new value of v_c is determined by:

$$upd_v\ (v_c, ss, is, v').$$

The initial value of the component of v_c that is addressed by *ss* and *is* is:

$$idx_v\ (sel_v\ (v_c, ss), is).$$

This sub-value is updated with v'. With Lemma 8.23 used from right to left, we can replace v' by the sequence of updates:

$$upd_v(v_c, ss, is, foldl\ (\lambda v\ ss'.\ upd_v(v, ss', [], sel_v(v',ss')))\ (idx_v(sel_v(v_c,ss),is))\ (selectors\ T)).$$

The sequence of updates is nested in the outer upd_v. The next step is to bring the *foldl* in front of the outer upd_v, so that the whole update is replaced by a sequence of updates. The following lemma allows to flatten the nested upd_v for a single step of this sequence.

Lemma 8.24 ► *If* $\vdash_v v_c :: T$ *and* $sel_T\ (T, ss) = \lfloor sT \rfloor$ *and* $idx_T\ (sT, is) = \lfloor iT \rfloor$ *and* $sel_T\ (iT, ss') = \lfloor sT' \rfloor$ *and* $idxfits\ (sel_v\ (v_c, ss), is)$ *and* $\vdash_v v' :: sT'$ *then*
$upd_v\ (v_c, ss, is, upd_v\ (idx_v\ (sel_v\ (v_c, ss), is), ss', [], v')) = upd_v\ (v_c, ss\ @\ ss', is, v')$.

Proof. By induction on the recursion-scheme of $sel_T\ (T, ss)$. □

In this lemma v' plays the role of $sel_v\ (v', ss')$ in the explanation above. The *ss* and the *is* are from the left expression, whereas the ss' is from splitting the value v'. The following lemma lifts this single step to a sequence of updates via *foldl* on a list *sss* of selector paths, and also transforms the update of the cell-value, to a state update via upd_c.

Lemma 8.25 ► *If* $\vdash_v dval_c\ T\ L\ c\ s :: T$ *and* $sel_T\ (T, ss) = \lfloor sT \rfloor$ *and* $idx_T\ (sT, is) = \lfloor iT \rfloor$ *and* $\vdash_v v' :: iT$ *and* $idxfits\ (sel_v\ (dval_c\ T\ L\ c\ s, ss), is)$ *and* $\neg\ null_c\ c$ *and* $\forall ss' \in set\ sss.\ sel_T\ (iT, ss') \neq None$ *and* $sss \neq []$ *then*

$$\begin{aligned}&upd_c\ L\ c\\ &\ (upd_v\ (dval_c\ T\ L\ c\ s, ss, is,\\ &\qquad foldl\ (\lambda v\ ss'.\ upd_v\ (v, ss', [], sel_v\ (v', ss')))\ (idx_v\ (sel_v\ (dval_c\ T\ L\ c\ s, ss), is))\ sss))\\ &\ s =\\ &foldl\ (\lambda s\ ss'.\ upd_c\ L\ c\ (upd_v\ (dval_c\ T\ L\ c\ s, ss\ @\ ss', is, sel_v\ (v', ss')))\ s)\ s\ sss.\end{aligned}$$

Proof. By induction on *sss* and Lemma 8.24. The restriction $\neg$ *null*$_c$ *c* is used in order to apply Lemma 8.15 to extract the sub-component of the cell-value that is actually changed by the outer *upd*$_v$. Note that *ss* and *is* remain the same in each step, only *ss'* varies. □

If we insert *selectors iT* for *sss* in Lemma 8.25, we can use Lemma 8.23 to collapse the complete inner *foldl* on the components of v' in the left hand side of the equation to v'. Moreover, we use Definition 8.40 of *upd* and arrive at the following lemma.

◄ Lemma 8.26

If $\vdash_v$ *dval*$_c$ *T L c s* :: *T and sel*$_T$ (*T*, *ss*) = ⌊*sT*⌋ *and idx*$_T$ (*sT*, *is*) = ⌊*iT*⌋ *and* $\vdash_v$ *v'* :: *iT and idxfits* (*sel*$_v$ (*dval*$_c$ *T L c s*, *ss*), *is*) *and distinct-selectors iT and* $\neg$ *null*$_c$ *c then*
upd T L (*LVal c ss is*) *v' s* =
foldl (λ*s ss'*. *upd T L* (*LVal c* (*ss* @ *ss'*) *is*) (*sel*$_v$ (*v'*, *ss'*)) *s*) *s* (*selectors iT*).

This lemma is the heart of the simulation of a C0 assignment *Ass le e* in Simpl. *LVar c ss is* corresponds to the reduced left expression *le*: *lval pn* [] [] *le* = *LVal c ss is*. Evaluation of *e* yields value v'. The left hand side of the equation corresponds to *assign L s le v* (cf. Lemma 8.17), and the right hand side corresponds to the sequence of assignments of the atomic components of v' in Simpl (cf. Lemma 8.16 and Definition 8.33). There is only one issue left. Among the premises of Lemma 8.16 there are three that depend on the state:

- definedness of the current cell: *val*$_c$ *L c s* = ⌊v_c⌋,
- type of the current cell value: $\vdash_v v_c$ :: *T*, and
- shape of the current cell value: *idxfits* (*sel*$_v$ (v_c, *ss*), *is*).

During the sequence of atomic updates the state changes as the cell is updated. Hence we have to show that these properties are invariant under *upd*. Update *upd* performs an *upd*$_v$ on value v_c. Hence definedness of v_c is preserved. Moreover, *upd*$_v$ preserves the type:

◄ Lemma 8.27

If HT$\vdash_v$ *v* :: *T and sel*$_T$ (*T*, *ss*) = ⌊*sT*⌋ *and idx*$_T$ (*sT*, *is*) = ⌊*iT*⌋ *and HT*$\vdash_v$ *v'* :: *iT and idxfits* (*sel*$_v$ (*v*, *ss*), *is*) *then*
HT$\vdash_v$ *upd*$_v$ (*v*, *ss*, *is*, *v'*) :: *T*.

Proof. By induction on the recursion-scheme of *upd*$_v$. □

Moreover, *upd*$_v$ preserves the shape of a value. Every valid index for a sub-component is still valid after an update of another sub-component.

◄ Lemma 8.28

If $\vdash_v$ *v* :: *T and sel*$_T$ (*T*, *ss*) = ⌊*sT*⌋ *and idx*$_T$ (*sT*, *is*) = ⌊*iT*⌋ *and idxfits* (*sel*$_v$ (*v*, *ss*), *is*), *moreover have* $\vdash_v$ *v'* :: *iT and sel*$_T$ (*T*, *ss'*) = ⌊*sT'*⌋ *and idx*$_T$ (*sT'*, *is'*) = ⌊*iT'*⌋ *and idxfits* (*sel*$_v$ (*v*, *ss'*), *is'*) *then*
idxfits (*sel*$_v$ (*upd*$_v$ (*v*, *ss*, *is*, *v'*), *ss'*), *is'*).

Proof. By induction on the recursion-scheme of *upd*$_v$. □

Lifting the previous two lemmas to *upd* we arrive at the desired invariant for the update. Selector path *ss* together with index list *is* describe the left-value. Selector path *ss'* is a extension to a sub-component.

Lemma 8.29 ► *If $val_c\ L\ c\ s = \lfloor v_c \rfloor$ and $HT \vdash_v v_c :: T$ and $sel_T\ (T, ss) = \lfloor sT \rfloor$ and $idx_T\ (sT, is) = \lfloor iT \rfloor$ and $idxfits\ (sel_v\ (v_c, ss), is)$ and $HT \vdash_v v :: iT$ and $sel_T\ (iT, ss') = \lfloor siT \rfloor$ then*

$\exists v_c'.\ val_c\ L\ c\ (upd\ T\ L\ (LVal\ c\ (ss\ @\ ss')\ is)\ (sel_v\ (v, ss'))\ s) = \lfloor v_c' \rfloor \wedge$
$\quad HT \vdash_v v_c' :: T \wedge idxfits\ (sel_v\ (v_c', ss), is)$.

Finally we can prove the simulation of a C0 assignment *Ass le e* in Simpl. We have to guard both the left-expression *le* and the expression *e* with a guard for an arbitrary atomic component. This also guards the assignment itself. If these guards hold, then the C0 assignment via *assign-opt* is simulated by the Simpl assignment via *ass*:

$$\begin{tikzcd} s_a \arrow[r, "\text{Simpl assignment with } ass"] & s_a' \\ s \arrow[u, "abs_s\ pn"] \arrow[r, "\text{C0 assignment with } assign\text{-}opt"'] & s' \arrow[u, "abs_s\ pn"'] \end{tikzcd}$$

Since $typ\ e \preceq typ\ le$ both *le* and *e* have the same components. The following auxiliary lemmas for widening are proven by induction on the widening relation.

Lemma 8.30 ► *If $T \preceq T'$ then $selectors\ T = selectors\ T'$.*

Lemma 8.31 ► *If $T \preceq T'$ then $implode\ (T, vs) = implode\ (T', vs)$.*

Theorem 8.32 ► (in *update*) *Simulation of assignment*

For a conforming C0-state s: $TE \vdash s :: HT, LT\ pn \restriction_A, GT$, where $TE = tnenv\ \Pi$ and an abstract Simpl-state $s_a \in abs_s\ pn\ s$, given a left-expression le that is welltyped: $\Pi, GT\ {+}{+}\ LT\ pn, HT \vdash_l le \surd$, and also definitely assigned: $\mathcal{D}_l\ le\ L\ A$, with respect to $A \subseteq dom\ (lvars\ s)$ and $L = dom\ (LT\ pn)$, given an expressions e that is welltyped: $\Pi, GT\ {+}{+}\ LT\ pn, HT \vdash_e e \surd$ and definitely assigned: $\mathcal{D}_e\ e\ L\ A$, and let $typ\ e \preceq typ\ le$, given a selector path ss: $sel_T\ (typ\ le, ss) = \lfloor sT \rfloor$, to an atomic type sT: $atomic_T\ sT$, then we have:

case $assign\text{-}opt\ L\ s\ (leval\ L\ s\ le)\ (eval\ L\ s\ e)$ of
$None \Rightarrow \neg\ s_a \in\in guard_e\ L\ pn\ ss\ []\ le \sqcap guard_e\ L\ pn\ ss\ []\ e$
$\mid \lfloor s' \rfloor \Rightarrow s_a \in\in guard_e\ L\ pn\ ss\ []\ le \sqcap guard_e\ L\ pn\ ss\ []\ e \longrightarrow$
$\quad ass\ (typ\ le)\ L\ (abs_l\ L\ pn\ []\ []\ le\ s_a)\ (\lambda ss.\ abs_e\ L\ pn\ ss\ []\ e\ s_a)\ s_a \in abs_s\ pn\ s'$.

Proof. In case the assignment causes a runtime-fault, then either *leval*, *eval* or *assign* caused this runtime-fault. According to Theorems 8.7 and 8.11 runtime faults in either *leval* or *eval* are detected by the guard for *le* or *e*, respectively. Moreover, the reduced left-expression that results from *leval* is also protected by the guard of *le* according to Theorem 8.11. In case the left-expression is a plain variable access, then the *assign* cannot fail. Otherwise $\mathcal{D}_l\ le\ l\ A$ implies $\mathcal{D}_e\ le\ l\ A$ (cf. Lemma 8.22). Typing and definite assignment of *le* are propagated to the resulting reduced left-expression (cf. Lemmas 7.15, 7.11 and 7.12). Hence a runtime fault in the assignment is detected by the guard of *le* (cf. Lemma 8.19 and Theorem 8.7).

In case the assignment does not cause a runtime fault, and both guards hold for the Simpl state then the evaluation of the left-expression *le* and the expression *e* are simulated by abs_l and abs_e, respectively (cf. Theorems 8.11 and 8.7). The abstracted value of expression *e* is provided in an exploded form to *ass*. The guard for *e* protects all components (cf. Lemma 8.8). Hence every atomic component of expression *e* is correctly abstracted by abs_e (cf. Theorem 8.7).

Function *ass* makes a case distinction on predicate *linit* that tests whether the assignment is an initialisation of a local variable. In case a local variable is initialised, the correctness of the simulation is already guaranteed by the requirements on LV_u in locale *update* (cf. Definition 8.34(2)) and Theorem 8.9, which guarantees that the original value of e is reconstructed by imploding its components. Lemma 8.31 justifies that *ass* can safely use the selectors of *typ le* instead of *typ e* to implode the value.

Otherwise, if the assignment is not an initialisation of a local variable, function *ass* simulates the assignment by a sequence of updates to the atomic components, which is justified by Lemma 8.26. Lemmas 8.17 and 8.16 are used to connect the update *upd* of Lemma 8.26 to *assign*, on the C0 side, and to abs_u on the Simpl side. The preconditions of Lemma 8.16 are invariant under state updates *upd* (cf. Lemma 8.29) and can be discharged via Lemma 8.18. □

8.5 Execution

So far we can simulate C0 expressions and assignments in Simpl. These are already the major building blocks for the simulation of a C0 execution in Simpl. We still have to consider memory allocation. In the C0 state the heap configuration consists of two parts, the mapping from locations to values and a counter for the free memory. All locations that are mapped to *None* are considered to be "free" or "fresh". If a new object is created a fresh location is chosen for this object, and the object itself is initialised with default values.

In the Simpl state we cannot see from a (split) heap which references are free, since there is no *option* layer. Instead the state is extended with an allocation list, that explicitly stores the allocated references. The references in the allocation list correspond to the location that are not mapped to *None* in C0. A counter for free memory is used in Simpl as well.

We introduce a new locale *allocate* that fixes the abstractions F, F_u, Al and Al_u for lookup and update of the free memory counter and the allocation list. Note that Al_u not only updates the allocation list, but also initialises the heap with the default values. We could think of splitting these two steps and describe the initialisation as ordinary assignment to the heap. The problem is that we cannot properly express the intermediate state after updating the allocation list and before initialisation with the default values in terms of the C0 state. Since in C0 allocation information and the values in the heap are both stored in same mapping, we would have to map the new location to a value $\lfloor v \rfloor$, since the location is allocated but cannot tell how v has to look like before it is properly initialised. We work around this issue, by keeping the allocation atomic in the abstraction to Simpl.

locale *allocate* = *update* + ◄ Definition 8.41
fixes
 free memory:
 $F :: {}'s \Rightarrow nat$
 consume memory:
 $F_u :: nat \Rightarrow {}'s \Rightarrow {}'s$
 allocated references:
 $Al :: {}'s \Rightarrow ref\ list$
 allocate new reference:
 $Al_u :: tname \Rightarrow ref \Rightarrow {}'s \Rightarrow {}'s$

assumes

(1) F corresponds to free-heap:

$s_a \in abs_s\ pn\ s \Longrightarrow F\ s_a = \mathit{free\text{-}heap}\ s$

(2) F_u simulates an update of free-heap:

$F_u\ n\ `\ abs_s\ pn\ s \subseteq abs_s\ pn\ (s(\!|\mathit{free\text{-}heap} := n|\!))$

(3) Al corresponds to the domain of the heap:

$[\![\mathit{finite}\ (\mathit{dom}\ (\mathit{heap}\ s));\ s_a \in abs_s\ pn\ s]\!]$
$\Longrightarrow \mathit{set}\ (Al\ s_a) = \mathit{Rep\text{-}loc}\ `\ \mathit{dom}\ (\mathit{heap}\ s)$

(4) Al_u simulates an allocation of a new heap object:

$[\![\mathit{finite}\ (\mathit{dom}\ (\mathit{heap}\ s));\ TE\ tn = \lfloor T\rfloor;\ s_a \in abs_s\ pn\ s;\ r = \mathit{new}\ (\mathit{set}\ (Al\ s_a))]\!]$
$\Longrightarrow Al_u\ tn\ r\ s_a \in abs_s\ pn\ (s(\!|\mathit{heap} := \mathit{heap}\ s(\mathit{Abs\text{-}loc}\ r \mapsto \mathit{default\text{-}val}\ T)|\!))$

The auxiliary function *alloc* implements pointer allocation for Simpl.

Definition 8.42 ► (in *allocate*)

```
alloc :: vname set ⇒ lval ⇒ tname ⇒ ty ⇒ 's ⇒ 's
alloc L lv tn T $s_a$ ≡
if sizeof-type T ≤ F $s_a$
then let r = new (set (Al $s_a$));
         $s_1$ = $Al_u$ tn r $s_a$;
         $s_2$ = $F_u$ (F $s_a$ − sizeof-type T) $s_1$;
         l = Abs-loc r
     in $abs_u$ L lv (Prim (Addr l)) $s_2$
else $abs_u$ L lv (Prim Null) $s_a$
```

The function directly resembles pointer allocation in C0. First, we test if there is still enough memory left. If not, the null pointer is assigned to the left-value. Otherwise we first obtain a fresh reference via *new*. This directly corresponds to the location that *new-Addr* yields in C0 (cf. Definition 7.32 on p. 133). Then we initialise the heap with the default value for the pointer, decrement the free memory counter and assign the reference to the left-value. For the assignment we can directly use abs_u since a pointer is an atomic value.

The expressions of a C0 statement have to be guarded in Simpl. The guard generating functions like guard_e yield an optional state set as guard. The functions *guard* and *guardWhile* annotate the guard to a command or loop if necessary. The guard for a loop condition *b* has to hold initially and after every execution of the loop body.

Definition 8.43 ►

$\mathit{guard} :: 'f \Rightarrow 's\ \mathit{set\ option} \Rightarrow ('s,'p,'f)\ \mathit{com} \Rightarrow ('s,'p,'f)\ \mathit{com}$
$\mathit{guard}\ f\ g\ c \equiv \mathsf{case}\ g\ \mathsf{of}\ \mathit{None} \Rightarrow c \mid \lfloor g\rfloor \Rightarrow \mathit{Guard}\ f\ g\ c$

Definition 8.44 ►

$\mathit{guardWhile} :: 'f \Rightarrow 's\ \mathit{set\ option} \Rightarrow 's\ \mathit{bexp} \Rightarrow ('s,'p,'f)\ \mathit{com} \Rightarrow ('s,'p,'f)\ \mathit{com}$
$\mathit{guardWhile}\ f\ g\ b\ c \equiv$
$\mathsf{case}\ g\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{While}\ b\ c \mid \lfloor g\rfloor \Rightarrow \mathit{Guard}\ f\ g\ (\mathit{While}\ b\ (\mathit{Seq}\ c\ (\mathit{Guard}\ f\ g\ \mathit{Skip})))$

Before we define the abstraction abs_c of C0 commands we introduce the locale *execute* that fixes a few more abstraction functions to handle procedure calls. The C0 program is Π, the corresponding Simpl procedure environment is Γ. With *PE* we can retrieve the list of parameters of a procedure, and with *RT* its return type. Moreover, *Ret* simulates the return of a procedure call. Parameter *f* is the fault that is raised when a guard is violated. Finally abs_S lifts state abstraction to optional states.

locale *execute* = *allocate* + ◂ Definition 8.45

fixes

parameter environment:
$PE :: pname \Rightarrow vname\ list$

return type:
$RT :: pname \Rightarrow ty$

default fault:
$f :: 'f$

C0 program:
$\Pi :: ty\ prog$

Simpl procedure environment:
$\Gamma :: pname \rightharpoonup ('s,pname,'f)\ com$

extended state abstraction:
$abs_S :: pname \Rightarrow state\ option \Rightarrow ('s,'f)\ xstate\ set$

assumes

(1) The state abstraction respects context switches:
$abs_s\ pn\ s \subseteq abs_s\ qn\ (s(\!|lvars := empty|\!))$

(2) Ret simulates the return from a procedure:
$[\![s_a \in abs_s\ pn\ s;\ t_a \in abs_s\ qn\ t]\!]$
$\Longrightarrow Ret\ s_a\ t_a \in abs_s\ pn\ (t(\!|lvars := lvars\ s|\!))$

(3) PE yields the parameters of a procedure:
$plookup\ \Pi\ pn = \lfloor((pds, lds, rT), bdy)\rfloor \Longrightarrow PE\ pn = map\ fst\ pds$

(4) RT yields the return type of a procedure:
$plookup\ \Pi\ pn = \lfloor((pds, lds, rT), bdy)\rfloor \Longrightarrow RT\ pn = rT$

(5) LT yields the local type environment of a procedure:
$plookup\ \Pi\ pn = \lfloor((pds, lds, rT), bdy)\rfloor \Longrightarrow$
$LT\ pn = map\text{-}of\ (pds\ @\ lds\ @\ [(Res, rT)])$

(6) GT yields the global type environment:
$GT = map\text{-}of\ (gdecls\text{-}of\ \Pi)$

(7) TE corresponds to the declared types:
$TE = tnenv\ \Pi$

(8) Function abs_S extends abs_s to optional states:
$abs_S\ pn\ s \equiv \mathsf{case}\ s\ \mathsf{of}\ None \Rightarrow \{Fault\ f\} \mid \lfloor s\rfloor \Rightarrow Normal\ `\ abs_s\ pn\ s$

(9) Γ yields the abstracted procedure bodies:
$\Gamma\ pn = option\text{-}map\ (abs_c\ pn \circ pbody\text{-}of)\ (plookup\ \Pi\ pn)$

Abstraction $abs_c\ pn\ c$ of C0 statement c in context of procedure pn is defined in Figure 8.4. ◂ Definition 8.46 (in *execute*)

The C0 *Skip* statement is mapped to the Simpl *Skip* statement.

For the assignment *Ass le e* we guard both the left-expression *le* and the expression *e*. The left-value of *le* is obtained with abs_l and the atomic components of *e* with abs_e. The assignment is simulated by a *Basic* statement with the function *ass*.

```
$abs_c$ :: $pname \Rightarrow ty\ stmt \Rightarrow ('s, pname,'f)\ com$

$abs_c\ pn\ Skip = Skip$

$abs_c\ pn\ (Ass\ le\ e) =$
let $T = typ\ le;$
    $g = guard_e\ (dom\ (LT\ pn))\ pn\ (hd\ (selectors\ T))\ []\ le\ \sqcap$
         $guard_e\ (dom\ (LT\ pn))\ pn\ (hd\ (selectors\ T))\ []\ e;$
    $lv = abs_l\ (dom\ (LT\ pn))\ pn\ []\ []\ le;$
    $v = \lambda s\ ss.\ abs_e\ (dom\ (LT\ pn))\ pn\ ss\ []\ e\ s$
in $guard\ f\ g\ (Basic\ (\lambda s.\ ass\ T\ (dom\ (LT\ pn))\ (lv\ s)\ (v\ s)\ s))$

$abs_c\ pn\ (PAlloc\ le\ tn) =$
let $\lfloor T \rfloor = TE\ tn;$
    $g = guard_e\ (dom\ (LT\ pn))\ pn\ []\ []\ le;$
    $lv = abs_l\ (dom\ (LT\ pn))\ pn\ []\ []\ le$
in $guard\ f\ g\ (Basic\ (\lambda s.\ alloc\ (dom\ (LT\ pn))\ (lv\ s)\ tn\ T\ s))$

$abs_c\ pn\ (Comp\ c_1\ c_2) = Seq\ (abs_c\ pn\ c_1)\ (abs_c\ pn\ c_2)$

$abs_c\ pn\ (Ifte\ e\ c_1\ c_2) =$
let $g = guard_e\ (dom\ (LT\ pn))\ pn\ (hd\ (selectors\ (typ\ e)))\ []\ e;$
    $b = \{s.\ the\text{-}Bool_v\ (abs_e\ (dom\ (LT\ pn))\ pn\ []\ []\ e\ s)\}$
in $guard\ f\ g\ (Cond\ b\ (abs_c\ pn\ c_1)\ (abs_c\ pn\ c_2))$

$abs_c\ pn\ (Loop\ e\ c) =$
let $g = guard_e\ (dom\ (LT\ pn))\ pn\ (hd\ (selectors\ (typ\ e)))\ []\ e;$
    $b = \{s.\ the\text{-}Bool_v\ (abs_e\ (dom\ (LT\ pn))\ pn\ []\ []\ e\ s)\}$
in $guardWhile\ f\ g\ b\ (abs_c\ pn\ c)$

$abs_c\ pn\ (SCall\ vn\ qn\ ps) =$
let $g = guard_{es}\ (dom\ (LT\ pn))\ pn\ ps;$
    $vs = \lambda s.\ map\ (\lambda e.\ implode\ (typ\ e, map\ (\lambda ss.\ abs_e\ (dom\ (LT\ pn))\ pn\ ss\ []\ e\ s)\ (selectors\ (typ\ e))))\ ps;$
    $init = \lambda s.\ foldl\ (\lambda s'\ (n, v).\ V_u\ (dom\ (LT\ qn))\ qn\ n\ []\ []\ v\ s')\ s\ (zip\ (PE\ qn)\ (vs\ s));$
    $lv = LVal\ (Var\ pn\ vn)\ []\ [];$
    $res = \lambda t\ ss.\ V\ (dom\ (LT\ qn))\ qn\ Res\ ss\ t\ [];$
    $result = \lambda i\ t.\ Basic\ (ass\ (RT\ qn)\ (dom\ (LT\ pn))\ lv\ (res\ t))$
in $guard\ f\ g\ (call\ init\ qn\ Ret\ result)$

$abs_c\ pn\ (Return\ e) =$
let $\lfloor T \rfloor = LT\ pn\ Res;$
    $g = guard_e\ (dom\ (LT\ pn))\ pn\ (hd\ (selectors\ T))\ []\ e;$
    $lv = LVal\ (Var\ pn\ Res)\ []\ [];$
    $v = \lambda s\ ss.\ abs_e\ (dom\ (LT\ pn))\ pn\ ss\ []\ e\ s$
in $guard\ f\ g\ (Basic\ (\lambda s.\ ass\ T\ (dom\ (LT\ pn))\ lv\ (v\ s)\ s))$
```

Figure 8.4: Abstraction of statements

For pointer allocation *PAlloc le tn* the left expression is guarded and reduced to a left-value via abs_l. The allocation and the assignment is handled by *alloc*.

Sequential composition *Comp* c_1 c_2 is directly mapped to *Seq* in Simpl.

For the conditional *Ifte* e c_1 c_2 the expression e is guarded and transformed to the corresponding set of Simpl states via abs_e. Then it is mapped to *Cond* in Simpl.

Similarly, for *Loop e c* the expression *e* is guarded and transformed to a state set in the Simpl *While* statement.

For the procedure call *SCall vn qn ps* the evaluation of the parameters *ps* is guarded by $guard_{es}$ and translated to *call init qn Ret result* in Simpl. Parameter passing is encoded in *init*. It iterates variable update V_u over the list of parameter names and values. The parameter names are obtained from *PE qn*, the parameter values *vs* are obtained by imploding the atomic components of the parameters *ps* that are accessed via abs_e. Exiting the procedure is handled by function *Ret*. The procedure *result* is read from the result variable *Res* in callee *qn* via function *res*, and assigned to variable *vn* of the caller *pn* with the left-value *lv*.

The return statement *Return e* is just a syntactic variant of an assignment to the result variable *Res*. Hence its translation is analogous to the translation of an assignment.

Abstraction of statements with abs_c is executable with Isabelle's simplifier. Hence a C0 statement can be automatically translated to the corresponding Simpl statement.

As we can see in Figure 8.4 the statement structure of the C0 and the corresponding Simpl program are quite the same. The main building blocks are guarding and simulating expressions and assignments. Since we already have the simulation theorems for these parts the proof for the simulation theorem for statements is rather straightforward. Given a C0 execution $\Pi,L\vdash_{C0} \langle c,\lfloor s\rfloor\rangle \Rightarrow t$ and an initial Simpl state $s_a \in abs_s\ pn\ s$, we show

- that there is an execution of the corresponding Simpl program, and
- that for the execution of the corresponding Simpl program the final state is either the fault state or contained in $abs_S\ pn\ t$.

Formulated in this canonical way, we do not have to exploit that C0 is actually deterministic. Also keep in mind, that the Simpl program can potentially cause more runtime-faults than the original C0 program, depending on the implementation of arithmetic. That is why we explicitly allow the Simpl program to end up in the fault state, although the C0 program may not cause a runtime fault. In case no fault occurs we have the following commuting diagram:

$$\begin{tikzcd}
Normal\ s_a \arrow[r, "\text{execute } abs_C\ pn\ c \text{ in Simpl}"] & t_a \\
\lfloor s \rfloor \arrow[u, "abs_S\ pn"] \arrow[r, "\text{execute } c \text{ in C0}"'] & t \arrow[u, "abs_S\ pn"']
\end{tikzcd}$$

◄ Theorem 8.33
Simulation of execution

In context of a wellformed program Π*: wf-prog* Π*, given a conforming C0-state s:* $TE\vdash s :: HT,LT\ pn\restriction_A,GT$*, and an abstract Simpl-state* $s_a \in abs_s\ pn\ s$*, given a welltyped statement c:* $\Pi,GT\ {+}{+}\ LT\ pn,HT\vdash c\ \surd$ *that is definitely assigned:* $\mathcal{D}\ c\ L\ A$*, with respect to* $L = dom\ (LT\ pn)$ *and* $A \subseteq dom\ (lvars\ s)$ *then we have:*

given the C0 execution $\Pi,L\vdash_{C0} \langle c,\lfloor s\rfloor\rangle \Rightarrow t$ *then*
$(\exists t_a.\ \Gamma\vdash \langle abs_c\ pn\ c,Normal\ s_a\rangle \Rightarrow t_a)\ \wedge$
$(\forall t_a.\ \Gamma\vdash \langle abs_c\ pn\ c,Normal\ s_a\rangle \Rightarrow t_a \longrightarrow t_a = Fault\ f \vee t_a \in abs_S\ pn\ t)$.

Proof. By induction on the C0 execution.

Case *Skip* is trivial.

Case *Ass le e* is covered by Theorem 8.32 for assignments.

Case *PAlloc le tn* is handled by the requirements for F, F_u, Al and Al_u (cf. Definition 8.41) and the simulation Theorem 8.32 for assignments. There is only one subtlety. Following the naming of states in Definition 8.42 of function *alloc*, the guard for the left-value is on the state s_a whereas we need it for the state s_2 in order to apply Theorem 8.32. By induction on the left-value we prove that the guard still holds in state s_2.

Case *Comp* c_1 c_2 follows from the induction hypotheses. As for the other compositional statements Theorem 7.16 about the type soundness of C0 is used to propagate conformance of the initial state to the intermediate states.

Cases *Ifte e* c_1 c_2 and *Loop e c* are derived from the induction hypotheses together with Theorem 8.9 for the simulation of the condition *e*.

Case *SCall vn qn ps*. Evaluation of parameters *ps* is covered by Theorem 8.10 for expression lists. By the requirement $abs_s\ pn\ s \subseteq abs_s\ qn\ (s(\!|lvars := empty|\!))$ of locale *execute* (cf. Definition 8.45(1)) we can simulate the entry of the procedure. By induction on the parameter list and the requirement on LV_u for the initialisation of local variables (cf. Definition 8.34(2)) we can simulate parameter passing. The wellformedness of the program ensures that the procedure body is welltyped and definitely assigned so that the induction hypothesis can be applied. Returning from the procedure is handled by the requirement on *Ret* (cf. Definition 8.45(2)). Finally the assignment of the result variable is covered by Theorem 8.32. Note that the result is assigned to a plain variable and thus no guard is needed here.

Case *Return e* is covered by Theorem 8.32 for assignments. □

This simulation theorem of a C0 execution by the execution of the corresponding Simpl program allows to transfer Hoare triples for partial correctness from Simpl to C0. If we have proven a Hoare triple on the Simpl level then this also includes the behaviour of the original C0 program. Hence the specification can be transferred to the C0 level. To transfer total correctness properties we additionally have to care about termination.

8.6 Termination

If we have proven that the Simpl program terminates we want to conclude that the original C0 program terminates, too. For both Simpl (cf. Figure 2.2 on p. 20) and C0 (cf. Figure 7.8 on p. 135) we have similar inductive characterisations of guaranteed termination. Hence the obvious proof idea is induction on the termination judgement for the Simpl program. We employ this idea but it does not work out as smoothly as one might expect. The basic problem is that we formally do induction on the termination of a Simpl program, but not on a general one, but one that was generated by abstracting a C0 program. So intuitively the induction is more on the termination of the embedded C0 program than on a general Simpl program.

We start with the terminating Simpl program $\Gamma \vdash c_a \downarrow Normal\ s_a$, where command $c_a = abs_c\ pn\ c$ for a C0 statement c and $s_a \in abs_s\ pn\ s$. We do induction on the termination judgement $\Gamma \vdash c_a \downarrow Normal\ s_a$ so in each inductive case we start with a Simpl program c_a and have to get hold of the C0 program c it comes from. So we have to invert the effect of abs_c. For example, in case $c_a = Seq\ c_{a1}\ c_{a2}$ we can do case analysis on c and see that only $c = \mathit{Comp}\ c_1\ c_2$ can result in c_a by applying abs_c. Hence we get both $c_{a1} = \mathit{abs}_c\ \mathit{pn}\ c_1$ and $c_{a2} = \mathit{abs}_c\ \mathit{pn}\ c_2$. We want to show that judgement $\Pi,L\vdash_{C0}\ \mathit{Comp}\ c_1\ c_2 \downarrow \lfloor s\rfloor$ holds. In this situation we have the following induction hypotheses:

- $\Pi,L\vdash_{C0}\ c_1 \downarrow \lfloor s\rfloor$ (∗)
- $\forall s_a{}'\ s'.\ \Gamma\vdash \langle c_{a1},\mathit{Normal}\ s_a\rangle \Rightarrow s_a{}' \longrightarrow s_a{}' \in \mathit{abs}_S\ \mathit{pn}\ s' \longrightarrow \Pi,L\vdash_{C0}\ c_2 \downarrow s'$. (∗∗)

According to the Comp Rule we have to show:

1. $\Pi,L\vdash_{C0}\ c_1 \downarrow \lfloor s\rfloor$
2. $\forall s'.\ \Pi,L\vdash_{C0}\ \langle c_1,\lfloor s\rfloor\rangle \Rightarrow s' \longrightarrow \Pi,L\vdash_{C0}\ c_2 \downarrow s'$.

We can already discharge (1) with hypothesis (∗). Also the conclusion of (2) matches the conclusion of (∗∗), but the preconditions are different. From (2) we get a C0 execution and have to transform it to a Simpl execution in order to discharge the preconditions of (∗∗). The simulation Theorem 8.33 for statements provides us with exactly this transformation. However, it also allows that the final state $s_a{}'$ of the Simpl execution can be the fault state $\mathit{Fault}\ f$, although s' is not None. In this case we cannot derive $s_a{}' \in \mathit{abs}_S\ \mathit{pn}\ s'$. Besides this dead end, the simulation theorem yields $s_a{}' \in \mathit{abs}_S\ \mathit{pn}\ s'$ and we can get hold of the conclusion of (∗∗). To get rid of the dead end we can restrict the Simpl executions to those that do not end up in a *Fault* state. In the end we want to transfer total correctness properties from Simpl to C0. The semantics of a Hoare triple (cf. Definition 3.2 on p. 39) guarantees that we do not end up in a *Fault* state. Therefore this is a perfectly legitimate restriction.

The example of the sequential composition shows that the basic proof idea seems to work and that we can build on the simulation theorem for C0 executions in Simpl. However, besides *Skip*, the only statement that never needs to be guarded is sequential composition. For all other statements we potentially get a guard in front of it. So instead of directly mapping *Ifte* to *Cond* we end up with $\mathit{Guard}\ f\ g\ (\mathit{Cond}\ \ldots)$. This destroys our inductive argument. Consider $c_a = \mathit{Guard}\ f\ g\ c_a{}'$ and $c_a = \mathit{abs}_c\ \mathit{pn}\ c$. In the example $c_a{}' = (\mathit{Cond}\ \ldots)$ and $c = (\mathit{Ifte}\ \ldots)$. We get an induction hypothesis for $c_a{}'$ but we cannot find any sub-statement c' of c that actually translates to $c_a{}'$, because guarding the statements is a crucial part of abs_c. To remedy the situation we consider termination of the Simpl skeletons, where all the guards are stripped off. Those skeletons preserve the structure of the original C0 abstract syntax and thus the induction hypothesis matches the subcomponents of a C0 statement. Is it legitimate to strip off the guards? Since runtime faults are also considered as termination, a program with guards potentially terminates more likely than the version without guards. In case we again restrict ourselves to executions of a guarded program that do not cause runtime faults, then the termination of the guarded program implies the termination of the program without guards (cf. Lemmas 5.7 and 5.8). This fits in our scenario, since total correctness of a program implies that no runtime error occurs.

However, even if we consider only the Simpl skeletons, without any guards, there is still one case where the induction hypothesis does not match: The procedure call. A C0 procedure call is translated to the derived command *call init p ret res* in Simpl (cf. Definition 2.8). It is defined in terms of *DynCom*. Now we get into a similar situation as with the guards. The procedure call is translated to $\mathit{DynCom}\ c_a{}'$, where $c_a{}'$ is defined according to Definition 2.8. We want to employ the induction hypothesis for the procedure body, but we only get one for $c_a{}'\ s_a$. Here $c_a{}'\ s_a$ has no proper C0 counterpart. Simpl and its termination judgement is too general for C0. C0 only exploits a subset of Simpl and hence the recursion and induction principles for Simpl do not suit to the Simpl image of C0. We regard the subset of Simpl statements that can be obtained by abs_c for a C0 statement as *C0-shaped*. For those statements we define a specialised termination judgement $\Gamma\vdash^{C0}_{shaped} c \downarrow s$ with the desired recursion structure. Rule induction on this judgement is sufficient to prove termination of the original C0 program. Moreover, we prove that every *C0-shaped* Simpl program that terminates with respect to the ordinary judgement $\Gamma\vdash c \downarrow s$ also terminates with respect to $\Gamma\vdash^{C0}_{shaped} c \downarrow s$. Ultimately, we can conclude that termination of the Simpl program, which was generated by abs_c implies that the original C0 program also terminates.

Definition 8.47 ► *The set of C0-shaped Simpl programs is defined inductively by the rules in 8.5.*

$$\frac{}{Skip \in C0\text{-}shaped} \qquad \frac{}{Basic\ f \in C0\text{-}shaped}$$

$$\frac{c_1 \in C0\text{-}shaped \qquad c_2 \in C0\text{-}shaped}{Seq\ c_1\ c_2 \in C0\text{-}shaped}$$

$$\frac{c_1 \in C0\text{-}shaped \qquad c_2 \in C0\text{-}shaped}{Cond\ b\ c_1\ c_2 \in C0\text{-}shaped} \qquad \frac{c \in C0\text{-}shaped}{While\ b\ c \in C0\text{-}shaped}$$

$$\frac{}{call\ init\ p\ ret\ (\lambda i\ t.\ Basic\ (f\ i\ t)) \in C0\text{-}shaped}$$

....

$$\frac{}{Call\ p \in C0\text{-}shaped} \qquad \frac{}{DynCom\ (\lambda t.\ Seq\ (Basic\ (f\ t))\ (Basic\ (g\ t))) \in C0\text{-}shaped}$$

$$\frac{c_1 \in C0\text{-}shaped \qquad c_2 \in C0\text{-}shaped}{Catch\ c_1\ c_2 \in C0\text{-}shaped} \qquad \frac{}{Throw \in C0\text{-}shaped}$$

Figure 8.5: C0-shaped Simpl programs

The rules above the dotted line are the obvious cases that result after stripping off the guards from the translation of a C0 statement with abs_c. Under the dotted line are the rules for those statements that appear as sub-statements of *call* if we expand Definition 2.8. This extension is needed for the proof of Lemma 8.35.

Stripping the guards after abstracting a C0 statement yields a *C0-shaped* result:

Lemma 8.34 ► (in *execute*)

$$strip\text{-}guards\ UNIV\ (abs_c\ pn\ c) \in C0\text{-}shaped$$

Proof. By induction on c. □

Definition 8.48 ► *Guaranteed termination $\Gamma\vdash^{C0}_{shaped} c \downarrow s$ of a C0-shaped program c in the initial state s is defined inductively by the rules in Figure 8.6, where:*

$$\begin{aligned}\Gamma &:: {}'p \rightharpoonup ({}'s, {}'p, {}'f)\ com\\ s &:: ({}'s, {}'f)\ xstate\\ c &:: ({}'s, {}'p, {}'f)\ com\end{aligned}$$

$$\frac{}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{Skip} \downarrow \mathit{Normal}\ s}\ (\textsc{Skip}) \qquad \frac{}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{Basic}\ f \downarrow \mathit{Normal}\ s}\ (\textsc{Basic})$$

$$\frac{\Gamma\vdash^{C0}_{\mathit{shaped}}\ c_1 \downarrow \mathit{Normal}\ s \qquad \forall s'.\ \Gamma\vdash \langle c_1,\mathit{Normal}\ s\rangle \Rightarrow s' \longrightarrow \Gamma\vdash^{C0}_{\mathit{shaped}}\ c_2 \downarrow s'}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{Seq}\ c_1\ c_2 \downarrow \mathit{Normal}\ s}\ (\textsc{Seq})$$

$$\frac{s \in b \qquad \Gamma\vdash^{C0}_{\mathit{shaped}}\ c_1 \downarrow \mathit{Normal}\ s}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{Cond}\ b\ c_1\ c_2 \downarrow \mathit{Normal}\ s}\ (\textsc{CondTrue}) \qquad \frac{s \notin b \qquad \Gamma\vdash^{C0}_{\mathit{shaped}}\ c_2 \downarrow \mathit{Normal}\ s}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{Cond}\ b\ c_1\ c_2 \downarrow \mathit{Normal}\ s}\ (\textsc{CondFalse})$$

$$\frac{s \in b \qquad \Gamma\vdash^{C0}_{\mathit{shaped}}\ c \downarrow \mathit{Normal}\ s \qquad \forall s'.\ \Gamma\vdash \langle c,\mathit{Normal}\ s\rangle \Rightarrow s' \longrightarrow \Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{While}\ b\ c \downarrow s'}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{While}\ b\ c \downarrow \mathit{Normal}\ s}\ (\textsc{WhileTrue})$$

$$\frac{s \notin b}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{While}\ b\ c \downarrow \mathit{Normal}\ s}\ (\textsc{WhileFalse})$$

$$\frac{\Gamma\ p = \lfloor \mathit{bdy} \rfloor \qquad \Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{bdy} \downarrow \mathit{Normal}\ (\mathit{init}\ s)}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{call}\ \mathit{init}\ p\ \mathit{ret}\ (\lambda i\ t.\ \mathit{Basic}\ (f\ i\ t)) \downarrow \mathit{Normal}\ s}\ (\textsc{CallPar})$$

$$\frac{\Gamma\ p = \mathit{None}}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{call}\ \mathit{init}\ p\ \mathit{ret}\ (\lambda i\ t.\ \mathit{Basic}\ (f\ i\ t)) \downarrow \mathit{Normal}\ s}\ (\textsc{CallParUndefined})$$

....

$$\frac{\Gamma\ p = \lfloor \mathit{bdy} \rfloor \qquad \Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{bdy} \downarrow \mathit{Normal}\ s}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{Call}\ p \downarrow \mathit{Normal}\ s}\ (\textsc{Call}) \qquad \frac{\Gamma\ p = \mathit{None}}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{Call}\ p \downarrow \mathit{Normal}\ s}\ (\textsc{CallUndefined})$$

$$\frac{}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{DynCom}\ (\lambda t.\ \mathit{Seq}\ (\mathit{Basic}\ (f\ t))\ (\mathit{Basic}\ (g\ t))) \downarrow \mathit{Normal}\ s}\ (\textsc{Return}) \qquad \frac{}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{Throw} \downarrow \mathit{Normal}\ s}\ (\textsc{Throw})$$

$$\frac{\Gamma\vdash^{C0}_{\mathit{shaped}}\ c_1 \downarrow \mathit{Normal}\ s \qquad \forall s'.\ \Gamma\vdash \langle c_1,\mathit{Normal}\ s\rangle \Rightarrow \mathit{Abrupt}\ s' \longrightarrow \Gamma\vdash^{C0}_{\mathit{shaped}}\ c_2 \downarrow \mathit{Normal}\ s'}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ \mathit{Catch}\ c_1\ c_2 \downarrow \mathit{Normal}\ s}\ (\textsc{Catch})$$

....

$$\frac{}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ c \downarrow \mathit{Fault}\ f}\ (\textsc{Fault}) \qquad \frac{}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ c \downarrow \mathit{Stuck}}\ (\textsc{Stuck}) \qquad \frac{}{\Gamma\vdash^{C0}_{\mathit{shaped}}\ c \downarrow \mathit{Abrupt}\ s}\ (\textsc{Abrupt})$$

Figure 8.6: Guaranteed termination of C0-shaped Simpl programs

The CallPar Rule is the one we are aiming at. It has the same recursion structure as a C0 procedure call. Hence induction on termination of C0-shaped programs fits well to the original C0 program. The rules between the dotted lines are again the extensions to the sub-statements occurring in the definition of *call* that we need for the following lemma. If a C0-shaped program terminates according to the original termination judgement, then it also terminates according to the new rules.

◂ Lemma 8.35

If $\Gamma\vdash c \downarrow s$ *and* $c \in \mathit{C0\text{-}shaped}$ *and* $\forall p\ \mathit{bdy}.\ \Gamma\ p = \lfloor \mathit{bdy} \rfloor \longrightarrow \mathit{bdy} \in \mathit{C0\text{-}shaped}$ *then* $\Gamma\vdash^{C0}_{\mathit{shaped}}\ c \downarrow s$.

Proof. By induction on $\Gamma\vdash c \downarrow s$ (cf. Figure 2.2 on p. 20). In case of $\mathit{DynCom}\ c_s$ we have the following hypotheses:

- $\Gamma\vdash c_s\ s \downarrow \mathit{Normal}\ s$,

- $DynCom\ c_s \in C0\text{-}shaped$, and (∗)
- $\Gamma\vdash c_s\ s \downarrow Normal\ s \longrightarrow c_s\ s \in C0\text{-}shaped \longrightarrow \Gamma\vdash^{C0}_{shaped} c_s\ s \downarrow Normal\ s.$ (∗∗)

To get hold of the conclusion of hypothesis (∗∗) we have to show that $c_s\ s$ is C0-shaped: $c_s\ s \in C0\text{-}shaped$. From (∗) we know that $DynCom\ c_s$ can either be of the form $call\ init\ p\ ret\ (\lambda i\ t.\ Basic\ (f\ i\ t))$ or it models a return statement, which results in the form $DynCom\ (\lambda t.\ Seq\ (Basic\ (f\ t))\ (Basic\ (g\ t)))$. In both cases we have $c_s\ s \in C0\text{-}shaped$.

The other cases of the induction are straightforward. □

Assuming that the Simpl program does not cause a runtime fault, and that the skeleton of the Simpl program, where all guards are stripped off, terminates according to the rules of C0-shaped termination, then the original C0 program terminates, too.

Lemma 8.36 ▶ (in *execute*) *In context of a wellformed program* Π*: wf-prog* Π*, given a conforming C0-state s:* $TE\vdash s :: HT,LT\ pn{\restriction_A},GT$*, and an abstract Simpl-state* $s_a \in abs_s\ pn\ s$*, given a welltyped statement c:* $\Pi,GT\ {+}{+}\ LT\ pn,HT\vdash c\ \surd$ *that is definitely assigned:* $\mathcal{D}\ c\ L\ A$*, with respect to* $A \subseteq dom\ (lvars\ s)$ *and* $L = dom\ (LT\ pn)$*, moreover let* $c_a = abs_c\ pn\ c$ *and let* $sc_a = strip\text{-}guards\ UNIV\ c_a$*, then we have:*

If strip UNIV $\Gamma\vdash^{C0}_{shaped} sc_a \downarrow Normal\ s_a$ *and* $\Gamma\vdash \langle c_a,Normal\ s_a\rangle \Rightarrow\notin Fault\ `\ UNIV \cup \{Stuck\}$ *then* $\Pi,L\vdash_{C0} c \downarrow \lfloor s\rfloor$*.*

Proof. By induction on $strip\ UNIV\ \Gamma\vdash^{C0}_{shaped} sc_a \downarrow Normal\ s_a$. In each inductive step we first construct the matching original C0 statement c from the given case of sc_a. This is done by case analysis on c and simplification according to the definitions of abs_c and $strip\text{-}guards$. For atomic C0 statements termination is trivial. For compound statements the induction hypothesis for termination of sc_a exactly fits to the preconditions of the termination judgement for C0 programs. The simulation of the execution of C0 statement c by the Simpl statement $c_a = abs_c\ pn\ c$ is provided by Theorem 8.33. The further simulation of c_a by the stripped version $sc_a = strip\text{-}guards\ UNIV\ c_a$ is provided by Lemmas 5.1 and 5.2. These Lemmas are applicable since we have excluded executions of c_a that end up in a *Fault* state. We exclude *Stuck* states, too. Hence we know that during execution every procedure is defined. This is necessary, since C0 does not handle undefined procedures in the existing semantics (cf. Figure 7.7 on p. 133) and termination judgement (cf. Figure 7.8 on p. 135). Welltyped C0 programs of course never call an undefined procedure, but we have not proven any formal lemma about this and thus cannot ignore this case. However, remember that the semantics of Hoare-triples in Simpl already ensures that the execution does not end up in a *Stuck* state and hence we do not introduce any substantial restriction here. □

The previous lemma uses the termination of the C0-shaped guard-less skeleton in order to make the induction work. The following simulation theorem for termination lifts this result to ordinary termination of the abstracted C0 program.

Theorem 8.37 ▶ (in *execute*) *Simulation of termination* *In context of a wellformed program* Π*: wf-prog* Π*, given a conforming C0-state s:* $TE\vdash s :: HT,LT\ pn{\restriction_A},GT$*, and an abstract Simpl-state* $s_a \in abs_s\ pn\ s$*, given a welltyped statement c:* $\Pi,GT\ {+}{+}\ LT\ pn,HT\vdash c\ \surd$ *that is definitely assigned:* $\mathcal{D}\ c\ L\ A$*, with respect to* $A \subseteq dom\ (lvars\ s)$ *and* $L = dom\ (LT\ pn)$*:*

If $\Gamma\vdash abs_c\ pn\ c \downarrow Normal\ s_a$ *and* $\Gamma\vdash \langle abs_c\ pn\ c,Normal\ s_a\rangle \Rightarrow\notin Fault\ `\ UNIV \cup \{Stuck\}$ *then* $\Pi,L\vdash_{C0} c \downarrow \lfloor s\rfloor$*.*

Proof. Let $c_a = abs_c\ pn\ c$ and $sc_a = strip\text{-}guards\ UNIV\ c_a$. Hence we have

- $\Gamma\vdash c_a \downarrow Normal\ s_a$, and (∗)
- $\Gamma\vdash \langle c_a,Normal\ s_a\rangle \Rightarrow\notin Fault\ `\ UNIV \cup \{Stuck\}$. (∗∗)

In order to apply Lemma 8.36, we first transform assumptions (∗) and (∗∗) to $strip\ UNIV\ \Gamma\vdash^{C0}_{shaped}\ sc_a \downarrow Normal\ s_a$. With (∗) and (∗∗) we utilise Lemma 5.7 and obtain $\Gamma\vdash sc_a \downarrow Normal\ s_a$. Moreover, from (∗∗) and Lemmas 5.1 and 5.3 we obtain $\Gamma\vdash \langle sc_a,Normal\ s_a\rangle \Rightarrow\notin Fault\ `\ UNIV$. Together we get $strip\ UNIV\ \Gamma\vdash sc_a \downarrow Normal\ s_a$ by Lemma 5.8. From Lemma 8.34 we know that sc_a and all bodies in $strip\ UNIV\ \Gamma$ are *C0-shaped*. Thus we can conclude with Lemma 8.35 that the stripped variant also terminates: $strip\ UNIV\ \Gamma\vdash^{C0}_{shaped}\ sc_a \downarrow Normal\ s_a$. Now we apply Lemma 8.36 to finish the proof. □

In order to transfer partial and total correctness properties we do not have to provide the dual theorem: "guaranteed termination of C0 programs implies guaranteed termination of the corresponding Simpl program". If there is a terminating execution of the C0 program, according to the big-step semantics, then there is at least one terminating computation in the corresponding Simpl program. This is already guaranteed by Theorem 8.33. Hence the behaviour of the C0 program is properly simulated by the corresponding Simpl program.

8.7 Hoare Triples

We can simulate C0 execution and termination in Simpl. Next we want to transfer program specifications, given in form of a Hoare triple, from the Simpl level back to the original C0 program. We first define the notion of a valid Hoare triple for C0 analogously to validity in Simpl, starting with partial correctness:

$$\Pi,L\models_{C0}\ P\ c\ Q \equiv \forall s\ t.\ \Pi,L\vdash_{C0} \langle c,s\rangle \Rightarrow t \longrightarrow s \in Some\ `\ P \longrightarrow t \in Some\ `\ Q$$

◂ Definition 8.49 *Validity (partial correctness)*

Given an execution of statement c from initial state s to final state t, provided that the initial state satisfy the precondition P then the execution of c does not cause a runtime fault and the final state satisfies the postcondition Q.

Total correctness additionally requires termination:

$$\Pi,L\models_{C0,t}\ P\ c\ Q \equiv \Pi,L\models_{C0}\ P\ c\ Q \wedge (\forall s{\in}Some\ `\ P.\ \Pi,L\vdash_{C0}\ c \downarrow s)$$

◂ Definition 8.50 *Validity (total correctness)*

In the corresponding Simpl specifications the postcondition for abrupt termination and the set of faults are both empty. We abbreviate $\Gamma\models_{/\{\}}\ P\ c\ Q,\{\}$ and also $\Gamma\models_{t/\{\}}\ P\ c\ Q,\{\}$ with $\Gamma\models P\ c\ Q$ and $\Gamma\models_t P\ c\ Q$, respectively. We want to transfer a Simpl Hoare triple $\Gamma\models P_a\ (abs_c\ pn\ c)\ Q_a$ to its C0 variant $\Pi,L\models_{C0}\ P\ c\ Q$. The assertions P_a and P as well as Q_a and Q have to be related. We introduce two ways to describe this relation, either by abstraction of a C0 assertion to a Simpl assertion, or by concretising a Simpl assertion to a C0 one. The differences between both approaches is discussed in this section.

Every state s for which there is an abstract state $s_a = abs_a\ pn\ s$ that satisfies P_a satisfies the concretisation of assertion P_a:

$$\begin{array}{l} concr :: pname \Rightarrow {}'s\ set \Rightarrow state\ set \\ concr\ pn\ P_a \equiv \{s.\ \exists s_a.\ s_a \in abs_s\ pn\ s \wedge s_a \in P_a\} \end{array}$$

◂ Definition 8.51 (in *execute*)

The union of the abstract states for all C0 states satisfying P is the abstraction of assertion P:

Definition 8.52 ▶ (in *execute*)

$$abs_a :: pname \Rightarrow state\ set \Rightarrow 's\ set$$
$$abs_a\ pn\ P \equiv \bigcup abs_s\ pn\ `\ P$$

The simulation Theorem 8.33 for C0 executions in Simpl only works for wellformed programs and welltyped definitely assigned statements and conforming states. Conformance and definite assignment both depend on the initial state. These properties have to be ensured by the precondition of the C0 Hoare triple. Welltypedness of the statement can also be established under the precondition of the Hoare triple. Remember that typing requires that all literal values are bounded. Since a specification may contain literal values, for example, as place-holders for procedure parameters, the bounds of these values are part of the precondition. Hence we need the precondition in order to prove welltypedness of the statement.

Theorem 8.38 ▶ (in *execute*) *Transfer of partial correctness (I)*

For a wellformed program Π*:* $wf\text{-}prog\ \Pi$*, given a statement* c *that is welltyped:* $\forall s{\in}P.\ \Pi,GT\ {+}{+}\ LT\ pn,HT{\vdash}\ c\ \surd$*, and also definitely assigned:* $\mathcal{D}\ c\ L\ A$*, with respect to* $L = dom\ (LT\ pn)$*, let* $P = concr\ pn\ P_a$ *and* $Q = concr\ pn\ Q_a$*, moreover assume that* $\forall s{\in}P.\ TE{\vdash}\ s :: HT,LT\ pn{\restriction}_A,GT$ *and* $\forall s{\in}P.\ A \subseteq dom\ (lvars\ s)$*, then we have:*

If $\Gamma{\models}\ P_a\ (abs_c\ pn\ c)\ Q_a$ *then* $\Pi,L{\models}_{C0}\ P\ c\ Q$*.*

Proof. According to Definition 8.49 of validity we have to consider a C0 execution $\Pi,L{\vdash}_{C0}\ \langle c,\lfloor s\rfloor\rangle \Rightarrow T$ where $s \in P$. We have to show that $T \in Some\ `\ Q$. From concretisation (cf. Definition 8.51) of P_a we obtain an abstract state $s_a \in abs_s\ pn\ s$ where $s_a \in P_a$. From the simulation Theorem 8.33 we get the corresponding Simpl execution $\Gamma{\vdash}\ \langle abs_c\ pn\ c,Normal\ s_a\rangle \Rightarrow T_a$, where either $T_a = Fault\ f$ or $T_a \in abs_S\ pn\ T$ (*). Moreover, we know from validity of the Simpl Hoare triple that no runtime faults occur and that $T_a \in Normal\ `\ Q_a$. Hence there is a $t_a$ with $T_a = Normal\ t_a$ and $t_a \in Q_a$ and with (*) also a $t$ such that $T = \lfloor t\rfloor$ and $t_a \in abs_s\ pn\ t$. By Definition 8.51 we can conclude $T \in Some\ `\ Q$. □

The previous proof of property transfer via concretisation of the assertions is straightforward. What if we want to use abstraction abs_a instead? We start with $\Gamma{\models}\ (abs_a\ pn\ P)\ abs_c\ pn\ c\ (abs_a\ pn\ Q)$ and want to conclude $\Pi,L{\models}_{C0}\ P\ c\ Q$. If we try to adapt the proof above we encounter a problem right in the first step. Given a $s \in P$, Definition 8.52 is not sufficient to ensure that there is a state $s_a \in abs_s\ pn\ s$, since abs_s could yield the empty set. A proper abstraction function is not as misbehaved, but formally we have to exclude this situation here. Another problem occurs in the final step of the proof, when we have given a $t_a \in abs_a\ pn\ Q$ and also $t_a \in abs_s\ pn\ t$. We want to conclude that $t \in Q$, but again the definition of abs_a is not sufficient. From $t_a \in abs_a\ pn\ Q$ we only know that there is a t' such that $t_a \in abs_s\ pn\ t'$ and $t' \in Q$. Unfortunately, t' and t do not necessarily have to be the same state. The underlying problem is that postcondition Q can potentially distinguish t' from t, whereas those differences are lost by the abstraction to t_a. We can restrict ourselves to postconditions Q that do not distinguish between states that are mapped to the same abstract state:

$$\forall t_1\ t_2.\ t_1 \in Q \longrightarrow abs_s\ pn\ t_1 \cap abs_s\ pn\ t_2 \neq \{\} \longrightarrow t_2 \in Q$$

Note that this issue does not occur with concretisation. There we start with an assertion on abstract states. Properties that can distinguish concrete states and not abstract ones, are thus *per se* ruled out since they cannot be expressed as assertion on abstract states.

◂ Theorem 8.39 (in *execute*) *Transfer of partial correctness (II)*

For a wellformed program Π*:* $\mathit{wf\text{-}prog}\ \Pi$*, given a statement* c *that is welltyped:* $\forall s{\in}P.\ \Pi,GT \mathbin{++} LT\ pn,HT\vdash c\ \surd$*, and also definitely assigned:* $\mathcal{D}\ c\ L\ A$*, with respect to* $L = \mathit{dom}\ (LT\ pn)$*, moreover assume that* $\forall s{\in}P.\ TE\vdash s :: HT,LT\ pn{\restriction_A},GT$ *and also* $\forall s{\in}P.\ A \subseteq \mathit{dom}\ (\mathit{lvars}\ s)$ *and* $\forall s{\in}P.\ \mathit{abs}_s\ pn\ s \neq \{\}$ *and that* Q *respects the state abstraction:* $\forall t_1\ t_2.\ t_1 \in Q \longrightarrow \mathit{abs}_s\ pn\ t_1 \cap \mathit{abs}_s\ pn\ t_2 \neq \{\} \longrightarrow t_2 \in Q$*, then we have:*
If $\Gamma\models (\mathit{abs}_a\ pn\ P)\ (\mathit{abs}_c\ pn\ c)\ (\mathit{abs}_a\ pn\ Q)$ *then* $\Pi,L\models_{C0} P\ c\ Q$*.*

Proof. According to Definition 8.49 we consider a C0 execution $\Pi,L\vdash_{C0} \langle c,\lfloor s\rfloor\rangle \Rightarrow T$ where $s \in P$. We have to show that $T \in \mathit{Some}\ `\ Q$. Since $\mathit{abs}_s\ pn\ s \neq \{\}$ we can obtain an abstract state $s_a \in \mathit{abs}_s\ pn\ s$ where $s_a \in P_a$ via Definition 8.52. From the simulation Theorem 8.33 we get the corresponding Simpl execution $\Gamma\vdash \langle \mathit{abs}_c\ pn\ c,\mathit{Normal}\ s_a\rangle \Rightarrow T_a$, where either $T_a = \mathit{Fault}\ f$ or $T_a \in \mathit{abs}_S\ pn\ T$ $(*)$. Moreover, we know from validity of the Simpl Hoare triple that runtime faults are excluded and $T_a \in \mathit{Normal}\ `\ \mathit{abs}_a\ pn\ Q$. Hence there is a t_a with $T_a = \mathit{Normal}\ t_a$ and $t_a \in \mathit{abs}_a\ pn\ Q$ $(**)$ and with $(*)$ also a state t such that $T = \lfloor t\rfloor$ and $t_a \in \mathit{abs}_s\ pn\ t$ $(***)$. From $(**)$ and Definition 8.52 we obtain another state t' such that $t' \in Q$ and $t_a \in \mathit{abs}_s\ pn\ t'$. Together with $(***)$ and our restriction on postcondition Q we can conclude that $t \in Q$ and hence $T \in \mathit{Some}\ `\ Q$. □

In practice it turns out that a combination of both theorems works the best. For the precondition we use abstraction and for the postcondition concretisation, together with a consequence step. In practical applications all the assertions P, P_a and Q and Q_a are given and we show:

- $\mathit{abs}_a\ pn\ P \subseteq P_a$ and
- $\mathit{concr}\ pn\ Q_a \subseteq Q$.

Unfolding the definitions yields the following proof obligations:

- $s_a \in \mathit{abs}_s\ pn\ s \longrightarrow s \in P \longrightarrow s_a \in P_a$ and
- $t_a \in \mathit{abs}_s\ pn\ t \longrightarrow t_a \in Q_a \longrightarrow t \in Q$.

In both cases we obtain either $s_a \in \mathit{abs}_s\ pn\ s$ or $t_a \in \mathit{abs}_s\ pn\ t$ which we can exploit in order to transfer the assertions.

Since the state abstraction only works well for conforming states it is crucial to have this information. For the precondition the transfer theorems already require conformance. For the postcondition we can derive it from the type soundness Theorem 7.16 and similarly for definite assignment (cf. Theorem 7.13).

◂ Corollary 8.40 (in *execute*)

For a wellformed program Π*:* $\mathit{wf\text{-}prog}\ \Pi$*, given a statement* c *that is welltyped:* $\forall s{\in}P.\ \Pi,GT \mathbin{++} LT\ pn,HT\vdash c\ \surd$*, and also definitely assigned:* $\mathcal{D}\ c\ L\ A$*, with respect to* $L = \mathit{dom}\ (LT\ pn)$*, moreover assume that* $\forall s{\in}P.\ TE\vdash s :: HT,LT\ pn{\restriction_A},GT$ *and also* $\forall s{\in}P.\ A \subseteq \mathit{dom}\ (\mathit{lvars}\ s)$ *and* $\forall s{\in}P.\ \mathit{abs}_s\ pn\ s \neq \{\}$*, provided that* $\mathit{abs}_a\ pn\ P \subseteq P_a$ *and*

$$\begin{aligned}&\forall HT'.\ HT \subseteq_m HT' \longrightarrow\\ &\qquad \mathit{concr}\ pn\ Q_a\\ &\qquad \subseteq \{t.\ TE\vdash t :: HT',LT\ pn{\restriction_{(A\,\cup\,\mathcal{A}\,c)}},GT \longrightarrow A \cup L \cap \mathcal{A}\ c \subseteq \mathit{dom}\ (\mathit{lvars}\ t) \longrightarrow t \in Q\},\end{aligned}$$

then we have:
If $\Gamma\models P_a\ (\mathit{abs}_c\ pn\ c)\ Q_a$ *then* $\Pi,L\models_{C0} P\ c\ Q$*.*

Proof. Analogous to Theorems 8.38 and 8.39. The additional assumptions in order to derive Q from Q_a are obtained by Theorems 7.16 and 7.13. □

For total correctness we can derive exactly the same theorems.

Theorem 8.41 ► (in *execute*) *Transfer of total correctness (I)*

For a wellformed program Π: wf-prog Π, given a statement c that is welltyped: $\forall s{\in}P.\ \Pi,GT +\!\!+ LT\ pn,HT \vdash c\ \surd$, and also definitely assigned: $\mathcal{D}\ c\ L\ A$, with respect to $L = dom\ (LT\ pn)$, let $\forall s{\in}P.\ A \subseteq dom\ (lvars\ s)$ and $\forall s{\in}P.\ TE \vdash s :: HT,LT\ pn{\restriction_A},GT$, moreover assume that $P = concr\ pn\ P_a$ and $Q = concr\ pn\ Q_a$, then we have:

If $\Gamma \models_t P_a\ (abs_c\ pn\ c)\ Q_a$ then $\Pi,L \models_{C0,t} P\ c\ Q$.

Proof. We have to show partial correctness and termination for the C0 program. The transfer of partial correctness follows from Theorem 8.38. For termination we assume $s \in P$ and show $\Pi,L \vdash_{C0} c \downarrow \lfloor s \rfloor$. From concretisation (cf. Definition 8.51) of P we obtain an abstract state $s_a \in abs_s\ pn\ s$ where $s_a \in P_a$. With validity of the Simpl Hoare triple we have $\Gamma \vdash \langle abs_c\ pn\ c, Normal\ s_a\rangle \Rightarrow \notin Fault\ `\ UNIV \cup \{Stuck\}$ and also $\Gamma \vdash abs_c\ pn\ c \downarrow Normal\ s_a$. Thus the simulation Theorem 8.37 for termination ensures $\Pi,L \vdash_{C0} c \downarrow \lfloor s \rfloor$. □

Theorem 8.42 ► (in *execute*) *Transfer of total correctness (II)*

For a wellformed program Π: wf-prog Π, given a statement c that is welltyped: $\forall s{\in}P.\ \Pi,GT +\!\!+ LT\ pn,HT \vdash c\ \surd$, and also definitely assigned: $\mathcal{D}\ c\ L\ A$, with respect to $L = dom\ (LT\ pn)$, moreover assume that $\forall s{\in}P.\ TE \vdash s :: HT,LT\ pn{\restriction_A},GT$ and also $\forall s{\in}P.\ A \subseteq dom\ (lvars\ s)$ and $\forall s{\in}P.\ abs_s\ pn\ s \neq \{\}$ and that Q respects the state abstraction: $\forall s\ t.\ s \in Q \longrightarrow abs_s\ pn\ s \cap abs_s\ pn\ t \neq \{\} \longrightarrow t \in Q$, then we have:

If $\Gamma \models_t (abs_a\ pn\ P)\ (abs_c\ pn\ c)\ (abs_a\ pn\ Q)$ then $\Pi,L \models_{C0,t} P\ c\ Q$.

Proof. Analogous to Theorem 8.41. Non-emptiness of the state abstraction guarantees a proper initial state $s_a \in abs_s\ pn\ s$ where $s_a \in P_a$. □

Corollary 8.43 ► (in *execute*)

For a wellformed program Π: wf-prog Π, given a statement c that is welltyped: $\forall s{\in}P.\ \Pi,GT +\!\!+ LT\ pn,HT \vdash c\ \surd$, and also definitely assigned: $\mathcal{D}\ c\ L\ A$, with respect to $L = dom\ (LT\ pn)$, moreover assume that $\forall s{\in}P.\ TE \vdash s :: HT,LT\ pn{\restriction_A},GT$ and also $\forall s{\in}P.\ A \subseteq dom\ (lvars\ s)$ and $\forall s{\in}P.\ abs_s\ pn\ s \neq \{\}$, provided that $abs_a\ pn\ P \subseteq P_a$ and $\forall HT'.\ HT \subseteq_m HT' \longrightarrow$

$$\begin{aligned}&concr\ pn\ Q_a\\ &\subseteq \{t.\ TE \vdash t :: HT',LT\ pn{\restriction_{(A \cup \mathcal{A}\ c)}},GT \longrightarrow\\ &\qquad A \cup L \cap \mathcal{A}\ c \subseteq dom\ (lvars\ t) \longrightarrow t \in Q\},\end{aligned}$$

then we have:

If $\Gamma \models_t P_a\ (abs_c\ pn\ c)\ Q_a$ then $\Pi,L \models_{C0,t} P\ c\ Q$.

Proof. Analogous to Theorems 8.41 and 8.42. The additional assumptions in order to derive Q from Q_a are obtained by Theorems 7.16 and 7.13. □

8.8 Example

The purpose of this section is to show that we can indeed build a model for all the assumptions that we have collected in order to prove the property transfer theorems. Moreover, we illustrate that for a given C0 program we can define the locale parameters like state abstraction and lookup and update functions schematically, and elaborate how their requirements can proven in an automatic fashion. Hence this instantiation can be automated.

We consider a program that consist of two procedures, one to calculate the factorial and one to reverse a list in the heap. The global array is only introduced to explain how to deal with arrays:

```
struct list {
  int cont;
  struct list* next;
};

struct list arr[10];

unsigned int Fac(unsigned int n) {
 unsigned int m;
 if (n=0) {
   return 1
 } else {
   m = Fac (n - 1);
   return (n*m);
 }
}

struct list* Rev(struct list* p) {
 struct node* q,r;
 q = NULL;
 while (p != NULL) {
   r = p;
   p = p->next;
   r->next = q;
   q = r;
 }
 return q;
}
```

The body of the procedure `Fac` has the following C0 syntax tree:

Fac-C0-bdy ≡
Ifte (*BinOp equal* (*VarAcc "n" UnsgndT*) (*Lit* (*Prim* (*Unsgnd 0*)) *UnsgndT*) *Boolean*)
 (*Return* (*Lit* (*Prim* (*Unsgnd 1*)) *UnsgndT*))
 (*Comp* (*SCall "m" "Fac"*
            [*BinOp minus* (*VarAcc "n" UnsgndT*) (*Lit* (*Prim* (*Unsgnd 1*)) *UnsgndT*)
              UnsgndT])
   (*Return* (*BinOp times* (*VarAcc "n" UnsgndT*) (*VarAcc "m" UnsgndT*) *UnsgndT*)))

The C0 syntax tree of procedure `Rev` is the following:

Rev-C0-bdy ≡
Comp (*Ass* (*VarAcc "q"* (*Ptr "list"*)) (*Lit* (*Prim Null*) *NullT*))
 (*Comp* (*Loop* (*BinOp notequal* (*VarAcc "p"* (*Ptr "list"*)) (*Lit* (*Prim Null*) *NullT*)
                    Boolean)

```
            (Comp (Ass (VarAcc "r" (Ptr "list")) (VarAcc "p" (Ptr "list")))
             (Comp (Ass (VarAcc "p" (Ptr "list"))
                        (StructAcc
                          (Deref (VarAcc "p" (Ptr "list"))
                            (Struct [("cont", Integer), ("next", Ptr "list")]))
                          "next" (Ptr "list")))
               (Comp (Ass (StructAcc
                               (Deref (VarAcc "r" (Ptr "list"))
                                 (Struct [("cont", Integer), ("next", Ptr "list")]))
                               "next" (Ptr "list"))
                          (VarAcc "q" (Ptr "list")))
                 (Ass (VarAcc "q" (Ptr "list")) (VarAcc "r" (Ptr "list")))))))
  (Return (VarAcc "q" (Ptr "list"))))
```

The complete program Π consists of the type declaration for list structures, the declaration of the global array variable and the procedure definitions:

```
Π ≡ ([("list", Struct [("cont", Integer), ("next", Ptr "list")])],
      [("arr", Arr 10 (Struct [("cont", Integer), ("next", Ptr "list")]))],
      [("Fac", ([("n", UnsgndT)], [("m", UnsgndT)], UnsgndT), Fac-C0-bdy),
       ("Rev", ([("p", Ptr "list")], [("q", Ptr "list"), ("r", Ptr "list")], Ptr "list"),
        Rev-C0-bdy)])
```

Next we define the Simpl state space. The global variables consist of the heaps *cont* and *next*, the components *arr-cont* and *arr-next* of the global array, and of the auxiliary components *alloc* and *free*. For the local variables we introduce the record fields *m*, *n* and result variable Res_n for procedure `Fac`, and *p*, *q*, *r* and Res_r for procedure `Rev`. Since the result variables of the factorial and list reversal have different types in the Simpl model, we need to introduce two components. In general we can map local variables of different procedures to the same record field as long as they share the same type in the Simpl model.

	record *st* =
record *globals* =	*globals* :: *globals*
alloc :: *ref list*	*n* :: *nat*
free :: *nat*	*m* :: *nat*
cont :: *ref* ⇒ *int*	Res_n :: *nat*
next :: *ref* ⇒ *ref*	*p* :: *ref*
arr-cont:: *int list*	*q* :: *ref*
arr-next:: *ref list*	*r* :: *ref*
	Res_r :: *ref*

We define the procedures in Simpl. We use unbounded arithmetic in Simpl and hence we have to guard the arithmetic operations in the program. Unsigned integers are mapped to natural numbers. For natural numbers we have $0 - 1 = 0$ for the subtraction and hence we introduce a guard. The upper bound for unsigned integers is *un-int-ub*. For the list reversal procedure we guard against dereferencing null pointers.

```
procedures Fac (n|Res_n) =
IF n = 0 THEN Res_n := 1
ELSE {|1 ≤ n|}↦ m := CALL Fac(n − 1);
  {|n ∗ m < un-int-ub|}↦ Res_n := n ∗ m
FI

procedures Rev (p|Res_r) =
q := NULL;
WHILE p ≠ NULL
DO r := p;
    {|p ≠ NULL|}↦ p := p→next;
    {|r ≠ NULL|}↦ r→next := q;
    q := r
OD;
Res_r := q
```

Now we specify the procedures and prove them correct on the Simpl level. For the factorial we get:

$$\forall n.\ \Gamma\vdash \{\!|n \le 12|\!\}\ \mathsf{Res}_n := \textbf{CALL}\ \mathit{Fac}(n)\ \{\!|\mathsf{Res}_n = \mathit{fac}\ n|\!\}$$

The upper bound *12* ensures that the calculation does not cause an overflow. For the list reversal we prove:

$$\forall p\ Ps.\ \Gamma\vdash \{\!|\mathit{List}\ p\ \mathsf{next}\ Ps|\!\}\ \mathsf{Res}_r := \textbf{CALL}\ \mathit{Rev}(p)\ \{\!|\mathit{List}\ \mathsf{Res}_r\ \mathsf{next}\ (\mathit{rev}\ Ps)|\!\}$$

Before going into detail about the instantiation of the transfer theorem, we discuss the results that we obtain from the property transfer to the C0 level. For the factorial we get:

$$\begin{aligned}\forall HT\ n.\ \Pi,L\models_{C0}\ &\{s.\ TE\vdash s :: HT,empty,GT \wedge n \le 12\}\\ &SCall\ ''Res''\ ''Fac''\ [Lit\ (Prim\ (Unsgnd\ n))\ UnsgndT]\\ &\{s.\ lvars\ s\ ''Res'' = \lfloor Prim\ (Unsgnd\ (fac\ n))\rfloor\}\end{aligned}$$

The specification resembles the Simpl Hoare triple. The precondition additionally restricts the initial state to be a conforming state. This is required by the property transfer Corollary 8.40, since the correspondence of a Simpl state to a C0 state is only given for conforming states. However, note that the typing for local variables is *empty*, which means that we do not have to put any restrictions on the local variables. This is a desired and important property of the specification. It allows to reuse the specification from any calling context since it does not make assumptions on the types of the local variables. The local environment can be *empty*, since the specified procedure call only gets a literal value as parameter and hence does not read any local variable of the caller. With respect to conformance, this specification can be used from any calling context. However, there is another handicap. The result is assigned to *"Res"*, which is the result variable of procedure *"Fac"* itself. The local variables L are also the local variables of the factorial: $L = dom\ (LT\ ''Fac'')$. This is because the statement has to be welltyped. Of course we could (re-)import the specification for every calling point in a C0 program, as we then know which local variables are active and which is the actual variable we assign the result to. However, this is annoying. This is not a problem of the property transfer theorem but of how to specify procedures on the C0 level. We have to deal with the result variable. In Simpl we solved this problem by completely decoupling procedure calls and parameter/result passing. A procedure specification only specifies the parameterless procedure. For C0 we could use the same idea and specify the procedure body instead of the procedure call. Alternatively, we can derive an adaptation rule that allows to adapt a specification of a canonical procedure call (with literal values as parameters, and an assignment to the formal result parameter) to an actual procedure call:

$$\frac{\forall x{\in}set\ ps.\ isLit\ x \qquad \begin{array}{c}\Pi,L{\models_{C0}}\ P\ (SCall\ x\ p\ ps)\ Q\\ \forall t\ r.\ update\text{-}var\ L\ t\ x\ r \in Q \longrightarrow update\text{-}var\ L'\ t\ y\ r \in Q'\end{array}}{\Pi,L'{\models_{C0}}\ P\ (SCall\ y\ p\ ps)\ Q'}$$

where $isLit\ e = (\exists v\ T.\ e = Lit\ v\ T)$

This rule allows to adapt the set of local variables from L to L', and the result variable from x to y. The side-condition ensures that the postconditions fit together. State t is the state after returning from the procedure before assigning the result r. Assigning r to x yields a state in Q and assigning r to y a state in Q'. Given a concrete Q and Q', rewriting of the side-condition substitutes accesses to x in Q with r, and similarly accesses of y in Q' with r. Applied to our example:

$$update\text{-}var\ L\ t\ ''Res''\ r \in \{t.\ lvars\ t\ ''Res'' = \lfloor Prim\ (Unsignd\ (fac\ n))\rfloor\}$$

simplifies to

$$r = \lfloor Prim\ (Unsignd\ (fac\ n))\rfloor.$$

And similarly, if we have another calling context L' and variable $y \in L'$ then:

$$update\text{-}var\ L'\ t\ y\ r \in \{t.\ lvars\ t\ y = \lfloor Prim\ (Unsignd\ (fac\ n))\rfloor\}$$

also reduces to

$$r = \lfloor Prim\ (Unsignd\ (fac\ n))\rfloor.$$

And thus the specification can be adapted in the desired fashion.

For the list reversal we get the following specification on the C0 level:

$$\begin{array}{l}\forall HT\ p\ Ls.\\ \quad \Pi,L{\models_{C0}}\ \{s.\ TE{\vdash}\ s :: HT,empty,GT \wedge \lfloor HT\rfloor{\vdash_v}\ p :: Ptr\ ''list'' \wedge List_{C0}\ p\ (heap\ s)\ Ls\}\\ \qquad\qquad SCall\ ''Res''\ ''Rev''\ [Lit\ p\ (Ptr\ ''list'')]\\ \qquad\qquad \{s.\ List_{C0}\ (the\ (lvars\ s\ ''Res''))\ (heap\ s)\ (rev\ Ls)\}\end{array}$$

Here we have to ensure that the value p is a pointer value, which means either *Prim Null* or an address *Prim* (*Addr l*). In case of an address the location l has to be registered in the heap typing HT as a list. This is all ensured by the typing constraint $\lfloor HT\rfloor{\vdash_v}\ p :: Ptr\ ''list''$. Again this is necessary to ensure welltypedness of the procedure call. The core part of the specification is the adaptation of the *List* predicate to the C0 level:

Definition 8.53 ►

$$\begin{array}{lcl}List_{C0} :: val \Rightarrow (loc \rightharpoonup val) \Rightarrow loc\ list \Rightarrow bool\\ List_{C0}\ v\ h\ [] & = & v = Prim\ Null\\ List_{C0}\ v\ h\ (l{\cdot}ls) & = & v = Prim\ (Addr\ l) \wedge List_{C0}\ (sel_v\ (the\ (h\ l),\ [''next'']))\ h\ ls\end{array}$$

After this preview on the results of the property transfer from Simpl to C0, we return to our starting point. We have to instantiate the whole framework in order to access the simulation theorems. After describing this instantiation we come back to the examples for some further discussion.

State abstraction The core of the simulation is to relate the C0 state with the Simpl state. To specify this relation we define the auxiliary function *assoc* that takes a default value, an association list and a key. If the key is in the association list it returns the corresponding element, otherwise it returns the default value.

◀ Definition 8.54

$$\begin{array}{lcl} assoc :: {'b} \Rightarrow ({'a} \times {'b})\ list \Rightarrow {'a} \Rightarrow {'b} & & \\ assoc\ d\ []\ k & = & d \\ assoc\ d\ ((k', e) \cdot as)\ k & = & \text{if } k' = k \text{ then } e \text{ else } assoc\ d\ as\ k \end{array}$$

We define the state abstraction abs_s with two auxiliary functions, one for the global components and one for the local variables:

$$abs_s\ pn\ s \equiv abs\text{-}glob\ (heap\ s)\ (free\text{-}heap\ s)\ (gvars\ s) \cap abs\text{-}loc\ pn\ (lvars\ s).$$

For the global components we relate *alloc* to the domain of the heap and *free* to *free-heap*. Moreover, every split heap is related to the corresponding projection of the C0 heap. We can see two kinds of projections. First with function *the*, to get rid of the option layer. This means that we only care about defined values. Second the projections *the-*... to convert a C0 *val* to the corresponding Simpl value. This means that we only care about type conforming stores. The global array is mapped to its components in the Simpl state-record:

$$\begin{array}{l} abs\text{-}glob :: heap \Rightarrow nat \Rightarrow vars \Rightarrow st\ set \\ abs\text{-}glob\ h\ f\ vs \equiv \\ \{s_a.\ (finite\ (dom\ h) \longrightarrow set\ (alloc\ (globals\ s_a)) = Rep\text{-}loc\ `\ dom\ h) \wedge \\ \quad free\ (globals\ s_a) = f \wedge \\ \quad (\forall l.\ next\ (globals\ s_a)\ (Rep\text{-}loc\ l) = the\text{-}Ref\ (sel_v\ (the\ (h\ l), [''next'']))) \wedge \\ \quad (\forall l.\ cont\ (globals\ s_a)\ (Rep\text{-}loc\ l) = the\text{-}Intg_v\ (sel_v\ (the\ (h\ l), [''cont'']))) \wedge \\ \quad arr\text{-}cont\ (globals\ s_a) = map\ the\text{-}Intg_v\ (the\text{-}Arrv\ (sel_v\ (the\ (vs\ ''arr''), [''cont'']))) \wedge \\ \quad arr\text{-}next\ (globals\ s_a) = map\ the\text{-}Ref\ (the\text{-}Arrv\ (sel_v\ (the\ (vs\ ''arr''), [''next''])))\} \end{array}$$

For the local variables we use the procedure name *pn* to associate the local variables to the corresponding record fields. In our example there are procedures *''Fac''* and *''Rev''*. The record fields correspond to the variables with the same name. The result variable *''Res''* is mapped to Res_n for procedure *''Fac''* and to Res_r for procedure *''Rev''*:

$$\begin{array}{l} abs\text{-}loc :: pname \Rightarrow (vname \rightharpoonup val) \Rightarrow st\ set \\ abs\text{-}loc \equiv \\ assoc\ (\lambda vs.\ UNIV) \\ \ [(''Fac'', \\ \quad \lambda vs.\ \{s_a.\ (vs\ ''n'' \neq None \longrightarrow n\ s_a = the\text{-}Unsgnd_v\ (the\ (vs\ ''n''))) \wedge \\ \qquad\qquad (vs\ ''m'' \neq None \longrightarrow m\ s_a = the\text{-}Unsgnd_v\ (the\ (vs\ ''m''))) \wedge \\ \qquad\qquad (vs\ ''Res'' \neq None \longrightarrow Res_n\ s_a = the\text{-}Unsgnd_v\ (the\ (vs\ ''Res'')))\}), \\ \ (''Rev'', \\ \quad \lambda vs.\ \{s_a.\ (vs\ ''p'' \neq None \longrightarrow p\ s_a = the\text{-}Ref\ (the\ (vs\ ''p''))) \wedge \\ \qquad\qquad (vs\ ''q'' \neq None \longrightarrow q\ s_a = the\text{-}Ref\ (the\ (vs\ ''q''))) \wedge \\ \qquad\qquad (vs\ ''r'' \neq None \longrightarrow r\ s_a = the\text{-}Ref\ (the\ (vs\ ''r''))) \wedge \\ \qquad\qquad (vs\ ''Res'' \neq None \longrightarrow Res_r\ s_a = the\text{-}Ref\ (the\ (vs\ ''Res'')))\})] \end{array}$$

We only place constraints on the local variables if they are defined in the C0 state. If the local variable mapping is *empty* every name is mapped to *None*. Hence *empty* local variables can be related to any abstract state. Moreover for undefined procedure names every abstract state is valid. With this construction we immediately obtain the context switch property of locale *execute* (cf. Definition 8.45(1)):

$$abs_s\ pn\ s \subseteq abs_s\ qn\ (s(\!|lvars := empty|\!))$$.

To return from a procedure is defined as

$$Ret \equiv \lambda s_a\ t_a.\ s_a(\!|globals := globals\ t_a|\!).$$

The initial caller state is s_a and the final state of the procedure body is t_a. We propagate the global variables to the caller state. Since the constraints on the global and local components are strictly separated we have

$$\begin{aligned}&(s_a(\!|globals := globals\ t_a|\!) \in abs\text{-}glob\ (heap\ t)\ (free\text{-}heap\ t)\ (gvars\ t)) = \\ &(t_a \in abs\text{-}glob\ (heap\ t)\ (free\text{-}heap\ t)\ (gvars\ t))\end{aligned}$$

and

$$(s_a(\!|globals := g|\!) \in abs\text{-}loc\ pn\ s) = (s_a \in abs\text{-}loc\ pn\ s).$$

Hence we can show that *Ret* properly simulates the return of a procedure in C0, as required in locale *execute* (cf. Definition 8.45(2)):

$$[\![s_a \in abs_s\ pn\ s;\ t_a \in abs_s\ qn\ t]\!] \Longrightarrow s_a(\!|globals := globals\ t_a|\!) \in abs_s\ pn\ (t(\!|lvars := lvars\ s|\!))$$

Another important property of the state abstraction is that for every C0 state there is at least one corresponding Simpl state. This is a precondition of the property transfer theorems (cf. Theorem 8.38):

$$\exists s_a.\ s_a \in abs_s\ pn\ s$$

We can prove this property by splitting the state record to its components. For each component and each procedure there is at most one active constraint that can be read as (conditional) definition. For example, for component n we have $vs\ ''n'' \neq None \longrightarrow n\ s_a = the\text{-}Unsgnd_v\ (the\ (vs\ ''n''))$. Hence we can just define component $n\ s_a$ as $the\text{-}Unsgnd_v\ (the\ (vs\ ''n''))$. The same is true for the global components with some additional lemmas. For the *alloc* list we have to prove that for every finite set of allocated locations there is a corresponding list of references, which is done by induction on finite sets. The *next* heap can be defined by the function $\lambda r.\ the\text{-}Ref\ (sel_v\ (the\ (h\ (Abs\text{-}loc\ r)),\ [''next'']))$. For this definition the constraint in *abs-glob* holds, since $Abs\text{-}loc\ (Rep\text{-}loc\ x) = x$ holds. The reason why we do not use this definition in *abs-glob* as well is that the translation functions from C0 start from a location l and convert it to $Rep\text{-}loc\ l$ and this exactly matches to the specification in *abs-glob*. Moreover we do not have to exclude *NULL* since this is implicitly already ensured by $Rep\text{-}loc\ l$. We provide the conversion lemma $\exists f.\ \forall l.\ f\ (Rep\text{-}loc\ l) = g\ l$ that ensures that we can define a split heap component from its specification in *abs-glob*.

Declarations Next we define the auxiliary functions to retrieve the declaration information of program Π. It is obvious that the requirements of locale *execute* (cf. Definition 8.45) are met by these definition.

- Type environment:
  $TE :: vname \rightharpoonup ty$
  $TE \equiv map\text{-}of\ [(''list'', Struct\ [(''cont'', Integer), (''next'', Ptr\ ''list'')])]$

- Typing of global variables:
  $GT :: vname \rightharpoonup ty$
  $GT \equiv map\text{-}of\ [(''arr'', Arr\ 10\ (Struct\ [(''cont'', Integer), (''next'', Ptr\ ''list'')]))]$
- Typing of local variables:
  $LT :: pname \Rightarrow vname \rightharpoonup ty$
  $LT \equiv flatten_m\ (map\text{-}of\ [(''Fac'', LT\text{-}Fac), (''Rev'', LT\text{-}Rev)])$
  $LT\text{-}Fac \equiv map\text{-}of\ [(''n'', UnsgndT), (''m'', UnsgndT), (''Res'', UnsgndT)]$
  $LT\text{-}Rev \equiv$
  $map\text{-}of\ [(''p'', Ptr\ ''list''), (''q'', Ptr\ ''list''), (''r'', Ptr\ ''list''), (''Res'', Ptr\ ''list'')]$
- Procedure parameters :
  $PE :: pname \Rightarrow vname\ list$
  $PE\ pn \equiv map\ fst\ (fst\ (fst\ (the\ (plookup\ \Pi\ pn))))$
- Procedure return type:
  $RT :: pname \Rightarrow ty$
  $RT\ pn \equiv snd\ (snd\ (fst\ (the\ (plookup\ \Pi\ pn))))$
- Abstract program:
  $\Gamma :: pname \rightharpoonup (st, pname, bool)\ com$
  $\Gamma\ pn \equiv option\text{-}map\ (abs_c\ pn \circ pbody\text{-}of)\ (plookup\ \Pi\ pn)$

The auxiliary function *flatten*$_m$ turns a nested mapping in a mapping that depends on both keys.

$$flatten_m :: ('a \rightharpoonup ('b \rightharpoonup 'c)) \Rightarrow ('a \Rightarrow ('b \rightharpoonup 'c))$$
$$flatten_m\ m \equiv \lambda x\ y.\ \textsf{case}\ m\ x\ \textsf{of}\ None \Rightarrow None \mid \lfloor m' \rfloor \Rightarrow m'\ y$$

◀ Definition 8.55

State lookup The lookup function *LV* for local variables takes a procedure name, a variable name, a selector list, the abstract state and an index list and returns the component of the corresponding variable. For primitive values the only relevant selector list is the empty one and the index list is completely ignored. We use function *assoc* to build *LV*. Since the cases for undefined variables or components are irrelevant for the desired properties of *LV* we use *arbitrary* as default element.

```
LV :: pname ⇒ vname ⇒ fname list ⇒ st ⇒ nat list ⇒ val
LV ≡ assoc arbitrary
        [("Fac",
          assoc arbitrary
           [("n", assoc arbitrary [([], λs_a is. Prim (Unsgnd (n s_a)))]),
            ("m", assoc arbitrary [([], λs_a is. Prim (Unsgnd (m s_a)))]),
            ("Res", assoc arbitrary [([], λs_a is. Prim (Unsgnd (Res_n s_a)))])]),
         ("Rev",
          assoc arbitrary
           [("p", assoc arbitrary [([], λs_a is. Ref (p s_a))]),
            ("q", assoc arbitrary [([], λs_a is. Ref (q s_a))]),
            ("r", assoc arbitrary [([], λs_a is. Ref (r s_a))]),
            ("Res", assoc arbitrary [([], λs_a is. Ref (Res_r s_a))])])]
```

If we lookup a variable(-component) via *LV*, the value has to be the same as if we fetch this value from the local variables directly (cf. Definition 8.20(1)):

$$\begin{array}{l}[\![\mathit{lvars}\ s\ \mathit{vn} = \lfloor v\rfloor;\ \mathit{LT}\ \mathit{pn}\ \mathit{vn} = \lfloor T\rfloor;\ \vdash_v v :: T;\ \mathit{sel}_T\ (T, \mathit{ss}) = \lfloor sT\rfloor;\\ \mathit{atomic}_T\ sT;\ \mathit{idxfits}\ (\mathit{sel}_v\ (v, \mathit{ss}), \mathit{is});\ s_a \in \mathit{abs}_s\ \mathit{pn}\ s]\!]\\ \Longrightarrow \mathit{LV}\ \mathit{pn}\ \mathit{vn}\ \mathit{ss}\ s_a\ \mathit{is} = \mathit{idx}_v\ (\mathit{sel}_v\ (v, \mathit{ss}), \mathit{is})\end{array}$$

How can we prove this property automatically? The point to start with is the second premise $\mathit{LT}\ \mathit{pn}\ \mathit{vn} = \lfloor T\rfloor$. We only have to consider the declared variables of a procedure. We pick one procedure after the other and test the property for each declared variable. For example, let us select procedure *"Fac"* and variable *"n"* which has type *UnsgndT*. Hence we know that v is of the form *Unsgnd i* for some i, since $\vdash_v v :: \mathit{UnsgndT}$. Since *UnsgndT* is a primitive type the only relevant selector path and index list is the empty list. According to the definition of *LV* we have:

$$\mathit{LV}\ "\mathit{Fac}"\ "n"\ []\ s_a\ [] = \mathit{Prim}\ (\mathit{Unsgnd}\ (n\ s_a)).$$

Since $s_a \in \mathit{abs}_s\ \mathit{pn}\ s$ and we know $\mathit{lvars}\ s\ "n" = \lfloor v\rfloor$ from the assumptions, we also have:

$$n\ s_a = \mathit{the\text{-}Unsgnd}_v\ (\mathit{the}\ (\mathit{lvars}\ s\ "n")).$$

Moreover from the assumptions we derived that $\mathit{lvars}\ s\ "n" = \lfloor \mathit{Prim}\ (\mathit{Unsgnd}\ i)\rfloor$. Hence the destructors *the* and $\mathit{the\text{-}Unsgnd}_v$ cancel the constructors $\lfloor\_\rfloor$ and *Unsgnd* and we arrive at:

$$\mathit{LV}\ "\mathit{Fac}"\ "n"\ []\ s_a\ [] = \mathit{Prim}\ (\mathit{Unsgnd}\ i) = \mathit{idx}_v(\mathit{sel}_v(\mathit{Prim}\ (\mathit{Unsgnd}\ i),[]),[]).$$

To prove the overall requirement we have to inspect all valid instances of the preconditions. That means we have to look at all procedures, all the local variable declarations and all the reasonable selector paths and index lists. How can we systematically enumerate all these cases? We start with $\mathit{LT}\ \mathit{pn}\ \mathit{vn} = \lfloor T\rfloor$. The type environment is defined with *map-of* out of an association list. By induction on the list we can prove the following induction scheme for maps:

Lemma 8.44 ►

$$\mathit{list\text{-}all}\ (\lambda(x, v).\ P\ x\ v)\ \mathit{xs} \Longrightarrow \mathit{map\text{-}of}\ \mathit{xs}\ x = \lfloor v\rfloor \longrightarrow P\ x\ v$$

If we want to prove a property $P\ x\ v$ for any key x that is mapped to v we prove this property for all the pairs in the association list *xs*. The function *list-all* tests whether a predicate holds for all list elements.

Definition 8.56 ►

$$\begin{array}{lcl}\multicolumn{3}{l}{\mathit{list\text{-}all} :: ('a \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathit{list} \Rightarrow \mathit{bool}}\\ \mathit{list\text{-}all}\ P\ [] & = & \mathit{True}\\ \mathit{list\text{-}all}\ P\ (x\cdot \mathit{xs}) & = & P\ x \wedge \mathit{list\text{-}all}\ P\ \mathit{xs}\end{array}$$

A similar theorem can be derived for flattening a map with $\mathit{flatten}_m$ like in the definition for *LT*. With this approach we can systematically enumerate the procedures and the local variable declarations. We still need a method for the selector paths and the index lists. For the selector paths we already have the function *selectors* that enumerates all the selectors to the atomic components of a type. The correctness Lemma 8.1 is also available. For the index list we do not enumerate every valid index. It is sufficient to focus on the length of the index lists which corresponds to the dimension of a type. For each index in this list the constraint $\mathit{idxfits}\ (\mathit{sel}_v\ (v, \mathit{ss}), \mathit{is})$ ensures that the index is within the array bounds. This is enough to symbolically evaluate list lookup and update by Isabelle's simplifier. To get hold of the valid dimension we derive the following equations for predicate *dimfits* (cf. Definition 8.8):

$$\begin{array}{lcl} \textit{dimfits}\ (\textit{Arr}\ n\ T)\ \textit{is} &=& \textsf{case}\ \textit{is}\ \textsf{of}\ []\Rightarrow \textit{True}\ |\ i{\cdot}\textit{is}' \Rightarrow \textit{dimfits}\ T\ \textit{is}' \\ \textit{dimfits}\ \text{-}\ \textit{is} &=& \textsf{case}\ \textit{is}\ \textsf{of}\ []\Rightarrow \textit{True}\ |\ a{\cdot}\textit{list} \Rightarrow \textit{False} \end{array}$$

◂ Lemma 8.45

With these equations we can automatically perform case analysis on the index list by rewriting. The type we apply *dimfits* to is *sT*. The following two lemmas allow to derive *dimfits sT*, from $\vdash_v v :: T$, $selT\ (ss, T) = sT$ and $idxfits\ (sel_v\ (v, ss), is)$.

If $sel_T\ (T, ss) = \lfloor sT \rfloor$ *and* $\vdash_v v :: T$ *then* $\vdash_v sel_v\ (v, ss) :: sT$. ◂ Lemma 8.46

Proof. By induction on the recursion scheme of *selT*. □

If $idxfits\ (v, is)$ *and* $\vdash_v v :: T$ *then* $dimfits\ T\ is$. ◂ Lemma 8.47

Proof. By induction on $\vdash_v v :: T$. □

With all this setup we do not have to prove the requirement for *LV* directly but start with a "executable" version:

```
list-all
 (λ(pn, m).
   ∀vn T. m vn = ⌊T⌋ ⟶
          lvars s vn = ⌊v⌋ ⟶
          ⊢_v v :: T ⟶
          s_a ∈ abs_s pn s ⟶
          list-all
           (λss. ∀sT. sel_T (T, ss) = ⌊sT⌋ ⟶
                      dimfits sT is ⟶
                      idxfits (sel_v (v, ss), is) ⟶ LV pn vn ss s_a is = idx_v (sel_v (v, ss), is))
           (selectors T))
 [("Fac", LT-Fac), ("Rev", LT-Rev)]
```

Simplification of this goal results in two subgoals, one for *"Fac"* and one for *"Rev"*. The subgoal for the factorial is the following:

```
list-all
 (λ(vn, T).
    lvars s vn = ⌊v⌋ ⟶
    ⊢_v v :: T ⟶
    s_a ∈ abs_s "Fac" s ⟶
    list-all
     (λss. ∀sT. sel_T (T, ss) = ⌊sT⌋ ⟶
                dimfits sT is ⟶
                idxfits (sel_v (v, ss), is) ⟶ LV "Fac" vn ss s_a is = idx_v (sel_v (v, ss), is))
     (selectors T))
 [("n", UnsgndT), ("m", UnsgndT), ("Res", UnsgndT)]
```

Now we just execute this proof obligation by rewriting. What is then left to show is something like *Prim* (*Unsgnd* ($the\text{-}Unsgnd_v$ *v*)) = *v* or more general a pattern like *constructor* (*destructor v*) = *v*. To prove this we have to know that *v* is of the right shape, e.g. again of the form *v* = *constructor v′* for some v′. Then both sides of the equation are reduced to *v′*. To achieve this we exploit the typing constraint $\vdash_v v :: T$. We can either build up a set of lemmas like

$$\vdash_v v :: \mathit{UnsgndT} \Longrightarrow \mathit{Prim}\ (\mathit{Unsgnd}\ (\mathit{the\text{-}Unsgnd}_v\ v)) = v$$

or subsequently expand $\vdash_v v :: T$. Since type T is instantiated with a concrete type we can use the following derived equations:

$$\begin{array}{lll}
\vdash_v v :: \mathit{Boolean} & = & \exists b.\ v = \mathit{Prim}\ (\mathit{Bool}\ b) \\
\vdash_v v :: \mathit{Integer} & = & \exists i.\ v = \mathit{Prim}\ (\mathit{Intg}\ i) \wedge \mathit{int\text{-}lb} \le i \wedge i < \mathit{int\text{-}ub} \\
\vdash_v v :: \mathit{UnsgndT} & = & \exists i.\ v = \mathit{Prim}\ (\mathit{Unsgnd}\ i) \wedge i < \mathit{un\text{-}int\text{-}ub} \\
\vdash_v v :: \mathit{CharT} & = & \exists i.\ v = \mathit{Prim}\ (\mathit{Chr}\ i) \wedge \mathit{chr\text{-}lb} \le i \wedge i < \mathit{chr\text{-}ub} \\
\vdash_v v :: \mathit{Ptr}\ \mathit{tn} & = & v = \mathit{Prim}\ \mathit{Null} \vee (\exists l.\ v = \mathit{Prim}\ (\mathit{Addr}\ l)) \\
\vdash_v v :: \mathit{NullT} & = & v = \mathit{Prim}\ \mathit{Null} \\
\vdash_v v :: \mathit{Struct}\ \mathit{cnTs} & = & \exists \mathit{cnvs}. \\
 & & \quad v = \mathit{Structv}\ \mathit{cnvs} \wedge \\
 & & \quad \mathit{map}\ \mathit{fst}\ \mathit{cnvs} = \mathit{map}\ \mathit{fst}\ \mathit{cnTs} \wedge \\
 & & \quad (\forall (v, T) \in \mathit{set}\ (\mathit{zip}\ (\mathit{map}\ \mathit{snd}\ \mathit{cnvs})\ (\mathit{map}\ \mathit{snd}\ \mathit{cnTs})).\ \vdash_v v :: T) \\
\vdash_v v :: \mathit{Arr}\ n\ T & = & \exists \mathit{av}.\ v = \mathit{Arrv}\ \mathit{av} \wedge |\mathit{av}| = n \wedge (\forall v \in \mathit{set}\ \mathit{av}.\ \vdash_v v :: T)
\end{array}$$

With this whole setup the requirement on *LV* can be proven automatically, driven by rewriting with a tactic like *fastsimp* in Isabelle. The requirement for global variable lookup via *GV* (cf. Definition 8.20(2)) can be handled in the same fashion. Here is the definition of *GV* for our program:

$$\begin{array}{l}
\mathit{GV} :: \mathit{vname} \Rightarrow \mathit{fname\ list} \Rightarrow \mathit{st} \Rightarrow \mathit{nat\ list} \Rightarrow \mathit{val} \\
\mathit{GV} \equiv \mathit{assoc\ arbitrary} \\
\quad\quad [(''\mathit{arr}'', \\
\quad\quad\ \ \mathit{assoc\ arbitrary} \\
\quad\quad\ \ [([''\mathit{cont}''], \\
\quad\quad\quad \lambda s_a\ \mathit{is}.\ \textsf{case}\ \mathit{is}\ \textsf{of}\ [] \Rightarrow \mathit{Arrv}\ (\mathit{map}\ (\mathit{Prim} \circ \mathit{Intg})\ (\mathit{arr\text{-}cont}\ (\mathit{globals}\ s_a))) \\
\quad\quad\quad\quad\quad\quad \mid i{\cdot}\mathit{is} \Rightarrow \mathit{Prim}\ (\mathit{Intg}\ (\mathit{arr\text{-}cont}\ (\mathit{globals}\ s_a))_{[i]})), \\
\quad\quad\ \ ([''\mathit{next}''], \\
\quad\quad\quad \lambda s_a\ \mathit{is}.\ \textsf{case}\ \mathit{is}\ \textsf{of}\ [] \Rightarrow \mathit{Arrv}\ (\mathit{map}\ \mathit{Ref}\ (\mathit{arr\text{-}next}\ (\mathit{globals}\ s_a))) \\
\quad\quad\quad\quad\quad\quad \mid i{\cdot}\mathit{is} \Rightarrow \mathit{Ref}\ (\mathit{arr\text{-}next}\ (\mathit{globals}\ s_a))_{[i]})])]
\end{array}$$

In case of an empty index list the whole array component is returned, in case of an one element index list the corresponding element is selected. We do not have to care about any other dimension of the index list since we only have to deal with welltyped lookups.

The heap lookup function *H* gets the type name and the selector list to determine the component of the split heap. Moreover it takes a reference, a Simpl state and the index list to actually lookup the value in the heap component:

$$\begin{array}{l}
H :: \mathit{tname} \Rightarrow \mathit{fname\ list} \Rightarrow \mathit{ref} \Rightarrow \mathit{st} \Rightarrow \mathit{nat\ list} \Rightarrow \mathit{val} \\
H \equiv \mathit{assoc\ arbitrary} \\
\quad\quad [(''\mathit{list}'', \\
\quad\quad\ \ \mathit{assoc\ arbitrary} \\
\quad\quad\ \ [([''\mathit{cont}''], \lambda r\ s_a\ \mathit{is}.\ \mathit{Prim}\ (\mathit{Intg}\ (\mathit{cont}\ (\mathit{globals}\ s_a)\ r))), \\
\quad\quad\ \ \ ([''\mathit{next}''], \lambda r\ s_a\ \mathit{is}.\ \mathit{Ref}\ (\mathit{next}\ (\mathit{globals}\ s_a)\ r))])]
\end{array}$$

The requirement on *H* in locale *lookup* (cf. Definition 8.20(3)) is quite similar to the ones for *LV* and *GV*. The type environment *TE* plays the role of *LT* or *GT*. Starting with the declared types of the program, we can apply the same automation ideas as before.

State update Function LV_u performs a state update of an atomic component in a local variable. It gets the procedure name, the variable name, the selector and the index list and the new value as parameters:

$$
\begin{array}{l}
LV_u :: pname \Rightarrow vname \Rightarrow fname\ list \Rightarrow nat\ list \Rightarrow val \Rightarrow st \Rightarrow st \\
LV_u \equiv \\
assoc\ arbitrary \\
\ [(''Fac'', \\
\quad assoc\ arbitrary \\
\quad [(''n'', assoc\ arbitrary\ [([], \lambda is\ v\ s_a.\ s_a(\!|n := the\text{-}Unsgnd_v\ v|\!))]), \\
\quad\ (''m'', assoc\ arbitrary\ [([], \lambda is\ v\ s_a.\ s_a(\!|m := the\text{-}Unsgnd_v\ v|\!))]), \\
\quad\ (''Res'', assoc\ arbitrary\ [([], \lambda is\ v\ s_a.\ s_a(\!|Res_n := the\text{-}Unsgnd_v\ v|\!))])]), \\
\ (''Rev'', \\
\quad assoc\ arbitrary \\
\quad [(''p'', assoc\ arbitrary\ [([], \lambda is\ v\ s_a.\ s_a(\!|p := the\text{-}Ref\ v|\!))]), \\
\quad\ (''q'', assoc\ arbitrary\ [([], \lambda is\ v\ s_a.\ s_a(\!|q := the\text{-}Ref\ v|\!))]), \\
\quad\ (''r'', assoc\ arbitrary\ [([], \lambda is\ v\ s_a.\ s_a(\!|r := the\text{-}Ref\ v|\!))]), \\
\quad\ (''Res'', assoc\ arbitrary\ [([], \lambda is\ v\ s_a.\ s_a(\!|Res_r := the\text{-}Ref\ v|\!))])])]
\end{array}
$$

We require from LV_u that the update commutes with the corresponding update in the C0 state (cf. Definition 8.34(1)):

$$
\begin{array}{l}
[\![lvars\ s\ vn = \lfloor v_c \rfloor;\ LT\ pn\ vn = \lfloor T \rfloor;\ \vdash_v v_c :: T;\ sel_T\ (T, ss) = \lfloor sT \rfloor; \\
\ atomic_T\ sT;\ idx_T\ (sT, is) = \lfloor iT \rfloor;\ idxfits\ (sel_v\ (v_c, ss), is);\ \vdash_v v :: iT]\!] \\
\Longrightarrow LV_u\ pn\ vn\ ss\ is\ v\ `\ abs_s\ pn\ s \\
\qquad \subseteq abs_s\ pn\ (s(\!|lvars := lvars\ s(vn \mapsto upd_v\ (v_c, ss, is, v))|\!))
\end{array}
$$

Like for the lookup we have to check all local variable declarations of all procedures and all reasonable selector paths and index lists. We can use the same automation techniques to enumerate all the relevant cases. Let us examine the update of variable $''n''$ in procedure $''Fac''$. The conclusion of the property is defined as set inclusion. We can transform this conclusion and introduce an abstract state $s_a \in abs_s\ pn\ s$ and show:

$$LV_u\ pn\ vn\ ss\ is\ v\ s_a \in abs_s\ pn\ (s(\!|lvars := lvars\ s(vn \mapsto upd_v\ (v_c, ss, is, v))|\!)).$$

We start with the left hand side. Since the type of variable $''n''$ is primitive we have $ss = []$ and $is = []$. According to the definition of LV_u we have:

$$LV_u\ ''Fac''\ ''n''\ []\ []\ v\ s_a = s_a(\!|n := the\text{-}Unsgnd_v\ v|\!).$$

On the right hand side we have $upd_v\ (v_c, [], [], v) = v$ and hence we have to show:

$$s_a(\!|n := the\text{-}Unsgnd_v\ v|\!) \in abs_s\ pn\ (s(\!|lvars := lvars\ s(''n'' \mapsto v)|\!)).$$

On both sides the only component that changes is n or the local variable at position $''n''$, respectively. All the other components stay the same and we get the simulation directly from $s_a \in abs_s\ pn\ s$. Referring to the function abs_s or more precise $abs\text{-}loc$ we have to show $n\ s_a' = the\text{-}Unsgnd_v\ (the\ (lvars\ s'\ ''n''))$ for the updated states $s_a' = s_a(\!|n := the\text{-}Unsgnd_v\ v|\!)$ and $s' = s(\!|lvars := lvars\ s(''n'' \mapsto v)|\!)$. This simplifies to:

$$the\text{-}Unsgnd_v\ v = the\text{-}Unsgnd_v\ v.$$

Of course, for non atomic values we do not only have the simple equation $upd_v\ (v_c, [], [], v) = v$ like above, but also cases where we have to reason about the commutation of selection and update. However, as we are only concerned with welltyped values and the situation where all indexes fit to the bounds of an array, this strategy works out in those cases, too.

In locale *update* there is another requirement on LV_u concerning the initialisation of local variables (cf. Definition 8.34(2)). Since our example only involves primitive types, this is actually the same. In general this requirement is easier to show since it does not involve the update of subcomponents on the C0 side.

The update GV_u for global variables looks a bit more involved since we have the additional indirection to the global variables.

```
GV_u :: vname ⇒ fname list ⇒ nat list ⇒ val ⇒ st ⇒ st
GV_u ≡
assoc arbitrary
 [("arr",
   assoc arbitrary
    [(["cont"],
      λis v s_a.
        case is of
        [] ⇒ s_a(|globals := globals s_a(|arr-cont := map the-Intg_v (the-Arrv v)|)|)
        | i·is ⇒ s_a
           (|globals := globals s_a
              (|arr-cont := arr-cont (globals s_a)[i := the-Intg_v v]|)|)),
     (["next"],
      λis v s_a.
        case is of
        [] ⇒ s_a(|globals := globals s_a(|arr-next := map the-Ref (the-Arrv v)|)|)
        | i·is ⇒ s_a
           (|globals := globals s_a
              (|arr-next := arr-next (globals s_a)[i := the-Ref v]|)|))])]
```

In order to prove the requirement for GV_u (cf. Definition 8.34(3)) we can use the same approach as for LV_u.

Heap update H_u uses the type name and the selector list to determine the heap and then performs the update in that heap at the position specified by the reference and the index list.

```
H_u :: tname ⇒ fname list ⇒ ref ⇒ nat list ⇒ val ⇒ st ⇒ st
H_u ≡
assoc arbitrary
 [("list",
   assoc arbitrary
    [(["cont"],
      λr is v s_a. s_a
        (|globals := globals s_a(|cont := (cont (globals s_a))(r := the-Intg_v v)|)|)),
     (["next"],
      λr is v s_a. s_a
        (|globals := globals s_a(|next := (next (globals s_a))(r := the-Ref v)|)|))])]
```

Again the proof of the commutation requirement of H_u in locale *update* (cf. Definition 8.34(4)) uses the same automation techniques as the proof for LV_u. Since the domain of the heap is not changed by the update the requirements for the allocation list and the free heap counter, which are imposed by abs_s, are also preserved by the update.

Memory Management For the abstraction of memory management we define the following functions:

- Lookup of free memory counter:
  $F :: st \Rightarrow nat$
  $F\ s_a \equiv free\ (globals\ s_a)$

- Update of free memory counter:
  $F_u :: nat \Rightarrow st \Rightarrow st$
  $F_u\ n\ s_a \equiv s_a (\!| globals := globals\ s_a (\!| free := n |\!) |\!)$

- Lookup of allocation list:
  $Al :: st \Rightarrow ref\ list$
  $Al\ s_a \equiv alloc\ (globals\ s_a)$

- Allocation and initialisation:
  $Al_u :: tname \Rightarrow ref \Rightarrow st \Rightarrow st$
  $Al_u\ tn\ r\ s_a \equiv$
  $assoc\ arbitrary$
  $[(''list'', s_a$
  $\quad (\!| globals := globals\ s_a$
  $\qquad (\!| alloc := r{\cdot}alloc\ (globals\ s_a),$
  $\qquad\ cont := (cont\ (globals\ s_a))(r := the\text{-}Intg_v\ (default\text{-}val\ Integer)),$
  $\qquad\ next := (next\ (globals\ s_a))(r := the\text{-}Ref\ (default\text{-}val\ (Ptr\ tn))) |\!) |\!))]$
  tn

The requirements for these functions are collected in locale *allocate* (cf. Definition 8.41) and most of them follow directly from the definitions. For the property about allocation and initialisation we enumerate all the declared types and then rewriting takes care of the rest.

Guarded Arithmetic We follow the approach to use unbounded arithmetic in Simpl. Hence we have to introduce guards to watch against over- and underflows in the C0 program. We implement the functions U and B for unary and binary operations, and the corresponding guard generators U_g and B_g. We only adapt the semantics for ordinary arithmetic. For bit-level operations we keep the original C0 semantics. The definition for unary expressions is in Figure 8.8, and for binary expressions in Figure 8.7. The commutations properties between this guarded arithmetic and the C0 arithmetic, as required in locale *lookup* (cf. Definition 8.20) are proven by exhaustive case distinction.

The definition of the equality and inequality test in B (cf. Figure 8.7) introduces more cases as the definition in *apply-binop* (cf. Figure 7.3). For each type there is an extra equation in order to get rid of the constructors for *prim* in Simpl.

Note that this definition of guarded arithmetic and the corresponding simulation lemmas are independent of our examples. They can be used for any C0 program.

$$\mathit{B} {::}\ (\mathit{binop} \times \mathit{prim} \times \mathit{prim}) \Rightarrow \mathit{prim}$$

$$\begin{aligned}
&B\ (\mathit{equal}, \mathit{Bool}\ b_1, \mathit{Bool}\ b_2) &&= \mathit{Bool}\ (b_1 = b_2)\\
&B\ (\mathit{equal}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) &&= \mathit{Bool}\ (i_1 = i_2)\\
&B\ (\mathit{equal}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) &&= \mathit{Bool}\ (n_1 = n_2)\\
&B\ (\mathit{equal}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) &&= \mathit{Bool}\ (i_1 = i_2)\\[4pt]
&B\ (\mathit{notequal}, \mathit{Bool}\ b_1, \mathit{Bool}\ b_2) &&= \mathit{Bool}\ (b_1 \neq b_2)\\
&B\ (\mathit{notequal}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) &&= \mathit{Bool}\ (i_1 \neq i_2)\\
&B\ (\mathit{notequal}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) &&= \mathit{Bool}\ (n_1 \neq n_2)\\
&B\ (\mathit{notequal}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) &&= \mathit{Bool}\ (i_1 \neq i_2)\\[4pt]
&B\ (\mathit{plus}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) &&= \mathit{Intg}\ (i_1 + i_2)\\
&B\ (\mathit{plus}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) &&= \mathit{Unsgnd}\ (n_1 + n_2)\\
&B\ (\mathit{plus}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) &&= \mathit{Chr}\ (i_1 + i_2)\\[4pt]
&B\ (\mathit{minus}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) &&= \mathit{Intg}\ (i_1 - i_2)\\
&B\ (\mathit{minus}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) &&= \mathit{Unsgnd}\ (n_1 - n_2)\\
&B\ (\mathit{minus}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) &&= \mathit{Chr}\ (i_1 - i_2)\\[4pt]
&B\ (\mathit{times}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) &&= \mathit{Intg}\ (i_1 * i_2)\\
&B\ (\mathit{times}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) &&= \mathit{Unsgnd}\ (n_1 * n_2)\\
&B\ (\mathit{times}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) &&= \mathit{Chr}\ (i_1 * i_2)\\[4pt]
&B\ (\mathit{divides}, \mathit{Intg}\ i_1, \mathit{Intg}\ i_2) &&= \mathit{Intg}\ (i_1\ \mathit{div}\ i_2)\\
&B\ (\mathit{divides}, \mathit{Unsgnd}\ n_1, \mathit{Unsgnd}\ n_2) &&= \mathit{Unsgnd}\ (n_1\ \mathit{div}\ n_2)\\
&B\ (\mathit{divides}, \mathit{Chr}\ i_1, \mathit{Chr}\ i_2) &&= \mathit{Chr}\ (i_1\ \mathit{div}\ i_2)\\[4pt]
&B\ (\mathit{bop}, v_1, v_2) &&= \mathit{the}\ (\mathit{apply\text{-}binop}\ (\mathit{bop}, v_1, v_2))
\end{aligned}$$

$$B_g {::}\ (\mathit{binop} \times \mathit{ty} \times \mathit{ty}) \Rightarrow ({}'s \Rightarrow \mathit{prim}) \Rightarrow ({}'s \Rightarrow \mathit{prim}) \Rightarrow ({}'s\ \mathit{set})\ \mathit{option}$$

$$\begin{aligned}
&B_g\ (\mathit{plus}, \mathit{Integer}, \mathit{Integer}) &&= \lambda v_1\ v_2.\ \lfloor\{s_a.\ \mathit{in\text{-}range\text{-}int}\ (\mathit{the\text{-}Intg}\ (v_1\ s_a) + \mathit{the\text{-}Intg}\ (v_2\ s_a))\}\rfloor\\
&B_g\ (\mathit{plus}, \mathit{UnsgndT}, \mathit{UnsgndT}) &&= \lambda v_1\ v_2.\ \lfloor\{s_a.\ \mathit{in\text{-}range\text{-}un\text{-}int}\ (\mathit{the\text{-}Unsgnd}\ (v_1\ s_a) + \mathit{the\text{-}Unsgnd}\ (v_2\ s_a))\}\rfloor\\
&B_g\ (\mathit{plus}, \mathit{CharT}, \mathit{CharT}) &&= \lambda v_1\ v_2.\ \lfloor\{s_a.\ \mathit{in\text{-}range\text{-}chr}\ (\mathit{the\text{-}Chr}\ (v_1\ s_a) + \mathit{the\text{-}Chr}\ (v_2\ s_a))\}\rfloor\\[4pt]
&B_g\ (\mathit{minus}, \mathit{Integer}, \mathit{Integer}) &&= \lambda v_1\ v_2.\ \lfloor\{s_a.\ \mathit{in\text{-}range\text{-}int}\ (\mathit{the\text{-}Intg}\ (v_1\ s_a) - \mathit{the\text{-}Intg}\ (v_2\ s_a))\}\rfloor\\
&B_g\ (\mathit{minus}, \mathit{UnsgndT}, \mathit{UnsgndT}) &&= \lambda v_1\ v_2.\ \lfloor\{s_a.\ \mathit{the\text{-}Unsgnd}\ (v_2\ s_a) \le \mathit{the\text{-}Unsgnd}\ (v_1\ s_a)\}\rfloor\\
&B_g\ (\mathit{minus}, \mathit{CharT}, \mathit{CharT}) &&= \lambda v_1\ v_2.\ \lfloor\{s_a.\ \mathit{in\text{-}range\text{-}chr}\ (\mathit{the\text{-}Chr}\ (v_1\ s_a) - \mathit{the\text{-}Chr}\ (v_2\ s_a))\}\rfloor\\[4pt]
&B_g\ (\mathit{times}, \mathit{Integer}, \mathit{Integer}) &&= \lambda v_1\ v_2.\ \lfloor\{s_a.\ \mathit{in\text{-}range\text{-}int}\ (\mathit{the\text{-}Intg}\ (v_1\ s_a) * \mathit{the\text{-}Intg}\ (v_2\ s_a))\}\rfloor\\
&B_g\ (\mathit{times}, \mathit{UnsgndT}, \mathit{UnsgndT}) &&= \lambda v_1\ v_2.\ \lfloor\{s_a.\ \mathit{in\text{-}range\text{-}un\text{-}int}\ (\mathit{the\text{-}Unsgnd}\ (v_1\ s_a) * \mathit{the\text{-}Unsgnd}\ (v_2\ s_a))\}\rfloor\\
&B_g\ (\mathit{times}, \mathit{CharT}, \mathit{CharT}) &&= \lambda v_1\ v_2.\ \lfloor\{s_a.\ \mathit{in\text{-}range\text{-}chr}\ (\mathit{the\text{-}Chr}\ (v_1\ s_a) * \mathit{the\text{-}Chr}\ (v_2\ s_a))\}\rfloor\\[4pt]
&B_g\ (\mathit{divides}, \mathit{Integer}, \mathit{Integer}) &&= \lambda v_1\ v_2.\ \lfloor\{s_a.\ \mathit{the\text{-}Intg}\ (v_2\ s_a) \neq 0 \wedge \mathit{the\text{-}Intg}\ (v_1\ s_a) \neq \mathit{int\text{-}lb}\}\rfloor\\
&B_g\ (\mathit{divides}, \mathit{UnsgndT}, \mathit{UnsgndT}) &&= \lambda v_1\ v_2.\ \lfloor\{s_a.\ \mathit{the\text{-}Unsgnd}\ (v_2\ s_a) \neq 0\}\rfloor\\
&B_g\ (\mathit{divides}, \mathit{CharT}, \mathit{CharT}) &&= \lambda v_1\ v_2.\ \lfloor\{s_a.\ \mathit{the\text{-}Chr}\ (v_2\ s_a) \neq 0 \wedge \mathit{the\text{-}Chr}\ (v_1\ s_a) \neq \mathit{chr\text{-}lb}\}\rfloor\\[4pt]
&B_g\ (\text{-},\text{-},\text{-}) &&= \lambda v_1\ v_2.\ \mathit{None}
\end{aligned}$$

$$\begin{aligned}
&\mathit{in\text{-}range\text{-}int}\ v &&= \mathit{int\text{-}lb} \le v \wedge v < \mathit{int\text{-}ub}\\
&\mathit{in\text{-}range\text{-}un\text{-}int}\ v &&= v < \mathit{un\text{-}int\text{-}ub}\\
&\mathit{in\text{-}range\text{-}chr}\ v &&= \mathit{chr\text{-}lb} \le v \wedge v < \mathit{chr\text{-}ub}
\end{aligned}$$

Figure 8.7: Guarded binary operations

$$
\begin{array}{ll}
\multicolumn{2}{l}{U :: (\mathit{unop} \times \mathit{prim}) \Rightarrow \mathit{prim}}\\
U\ (\mathit{unary\text{-}minus},\ \mathit{Intg}\ i) & = \mathit{Intg}\ (-\ i)\\
U\ (\mathit{unary\text{-}minus},\ \mathit{Chr}\ i) & = \mathit{Chr}\ (-\ i)\\
U\ (\mathit{to\text{-}int},\ \mathit{Intg}\ i) & = \mathit{Intg}\ i\\
U\ (\mathit{to\text{-}int},\ \mathit{Unsgnd}\ n) & = \mathit{Intg}\ (\mathit{int}\ n)\\
U\ (\mathit{to\text{-}int},\ \mathit{Chr}\ i) & = \mathit{Intg}\ i\\
U\ (\mathit{to\text{-}unsigned\text{-}int},\ \mathit{Intg}\ i) & = \mathit{Unsgnd}\ (\mathit{nat}\ i)\\
U\ (\mathit{to\text{-}unsigned\text{-}int},\ \mathit{Unsgnd}\ n) & = \mathit{Unsgnd}\ n\\
U\ (\mathit{to\text{-}unsigned\text{-}int},\ \mathit{Chr}\ i) & = \mathit{Unsgnd}\ (\mathit{nat}\ i)\\
U\ (\mathit{to\text{-}char},\ \mathit{Intg}\ i) & = \mathit{Chr}\ i\\
U\ (\mathit{to\text{-}char},\ \mathit{Unsgnd}\ n) & = \mathit{Chr}\ (\mathit{int}\ n)\\
U\ (\mathit{to\text{-}char},\ \mathit{Chr}\ i) & = \mathit{Chr}\ i\\
U\ (\mathit{uop},\ v) & = \mathit{the}\ (\mathit{apply\text{-}unop}\ (\mathit{uop},\ v))
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{U_g :: (\mathit{unop} \times \mathit{ty}) \Rightarrow ('s \Rightarrow \mathit{prim}) \Rightarrow ('s\ \mathit{set})\ \mathit{option}}\\
U_g\ (\mathit{unary\text{-}minus},\ \mathit{Integer}) & = \lambda v.\ \lfloor\{s_a.\ \mathit{int\text{-}lb} < \mathit{the\text{-}Intg}\ (v\ s_a)\}\rfloor\\
U_g\ (\mathit{unary\text{-}minus},\ \mathit{CharT}) & = \lambda v.\ \lfloor\{s_a.\ \mathit{chr\text{-}lb} < \mathit{the\text{-}Chr}\ (v\ s_a)\}\rfloor\\
U_g\ (\mathit{to\text{-}int},\ \mathit{UnsgndT}) & = \lambda v.\ \lfloor\{s_a.\ \mathit{the\text{-}Unsgnd}\ (v\ s_a) < \mathit{nat}\ \mathit{int\text{-}ub}\}\rfloor\\
U_g\ (\mathit{to\text{-}unsigned\text{-}int},\ \mathit{Integer}) & = \lambda v.\ \lfloor\{s_a.\ 0 \le \mathit{the\text{-}Intg}\ (v\ s_a)\}\rfloor\\
U_g\ (\mathit{to\text{-}unsigned\text{-}int},\ \mathit{CharT}) & = \lambda v.\ \lfloor\{s_a.\ 0 \le \mathit{the\text{-}Chr}\ (v\ s_a)\}\rfloor\\
U_g\ (\mathit{to\text{-}char},\ \mathit{Integer}) & = \lambda v.\ \lfloor\{s_a.\ \mathit{in\text{-}range\text{-}chr}\ (\mathit{the\text{-}Intg}\ (v\ s_a))\}\rfloor\\
U_g\ (\mathit{to\text{-}char},\ \mathit{UnsgndT}) & = \lambda v.\ \lfloor\{s_a.\ \mathit{the\text{-}Unsgnd}\ (v\ s_a) < \mathit{nat}\ \mathit{chr\text{-}ub}\}\rfloor\\
U_g\ (\text{-},\text{-}) & = \lambda v.\ \mathit{None}
\end{array}
$$

$$\mathit{in\text{-}range\text{-}chr}\ v\ =\ \mathit{chr\text{-}lb} \le v \wedge v < \mathit{chr\text{-}ub}$$

Figure 8.8: Guarded unary operations

Property Transfer We have defined all the locale parameters and have proven all the necessary properties. Hence we can instantiate the locale *execute* and Isabelle provides us with all the instances of the generic theorems for the current definitions. Let us reconsider the initial examples of the factorial and the list reversal procedure and examine more closely how the specifications are ported from Simpl to the C0. For the factorial we have the following Hoare triple in Simpl:

$$\forall n.\ \Gamma\vdash \{\!|n \le 12|\!\}\ \mathsf{Res}_n := \textbf{CALL}\ \mathit{Fac}(n)\ \{\!|\mathsf{Res}_n = \mathit{fac}\ n|\!\}$$

We can transfer it to the C0 variant:

$$
\begin{array}{l}
\forall \mathit{HT}\ n.\ \Pi,L\models_{C0} \{s.\ \mathit{TE}\vdash s :: \mathit{HT},\mathit{empty},\mathit{GT} \wedge n \le 12\}\\
\qquad\qquad\qquad \mathit{SCall}\ \text{"Res"}\ \text{"Fac"}\ [\mathit{Lit}\ (\mathit{Prim}\ (\mathit{Unsgnd}\ n))\ \mathit{UnsgndT}]\\
\qquad\qquad\qquad \{s.\ \mathit{lvars}\ s\ \text{"Res"} = \lfloor \mathit{Prim}\ (\mathit{Unsgnd}\ (\mathit{fac}\ n))\rfloor\}
\end{array}
$$

For list reversal the Simpl specification is:

$$\forall p\ \mathit{Ps}.\ \Gamma\vdash \{\!|\mathit{List}\ p\ \mathsf{next}\ \mathit{Ps}|\!\}\ \mathsf{Res}_r := \textbf{CALL}\ \mathit{Rev}(p)\ \{\!|\mathit{List}\ \mathsf{Res}_r\ \mathsf{next}\ (\mathit{rev}\ \mathit{Ps})|\!\}$$

And is transformed to:

$$
\begin{array}{l}
\forall HT\ p\ Ls.\\
\quad \Pi,L\models_{C0} \{s.\ TE\vdash s :: HT,empty,GT \wedge \lfloor HT\rfloor\vdash_v p :: Ptr\ ''list'' \wedge List_{C0}\ p\ (heap\ s)\ Ls\}\\
\qquad\qquad SCall\ ''Res''\ ''Rev''\ [Lit\ p\ (Ptr\ ''list'')]\\
\qquad\qquad \{s.\ List_{C0}\ (the\ (lvars\ s\ ''Res''))\ (heap\ s)\ (rev\ Ls)\}
\end{array}
$$

The theorem we use for this property transfer is Corollary 8.40. First, we have to prove wellformedness of program Π. All type and variable declarations have to be wellformed and all procedure bodies have to be welltyped and definitely assigned. All these tests are specified as syntax directed inductive definitions or recursive functions. Since the C0 expressions are annotated with types we do not need any kind of type inference. The built in automation of Isabelle is sufficient to prove wellformedness of the program automatically by a tactic like *fastsimp*.

The statements in the Hoare triples we aim to transfer are procedure calls, like *SCall ''Res'' ''Fac'' [Lit (Prim (Unsgnd n)) UnsgndT]*. This call is welltyped with respect to *LT ''Fac''*. Moreover, it passes the definite assignment analysis for any set of local variables L and set of assigned variables A, since the parameter is a literal value. Hence we can use $A = \{\}$. For the conformance restriction of the initial sate the emptiness of A also comes in handy: $LT\ ''Fac''\upharpoonright_A = empty$. This is a desirable effect since it means that the precondition is independent of the local variables of the caller.

We have already proven that $abs_s\ pn\ s \neq \{\}$ for any state s and procedure pn.

The translation from a C0 statement c to the Simpl statement $abs_c\ pn\ c$ is performed automatically by evaluation of abs_c with Isabelle's simplifier.

The only point left is the adaptation of the pre- and postconditions. We start with $abs_a\ pn\ P \subseteq P_a$. For the factorial we have $P = \{s.\ TE\vdash s :: HT,empty,GT \wedge n \leq 12\}$ and $P_a = \{\!|n \leq 12|\!\}$. Hence the relation between both assertions is trivial to prove, since P_a does not even depend on the state but only on the logical variable n. For the list reversal the situation is different. Here we have the C0 precondition $P = \{s.\ TE\vdash s :: HT,empty,GT \wedge \lfloor HT\rfloor\vdash_v p :: Ptr\ ''list'' \wedge List_{C0}\ p\ (heap\ s)\ Ls\}$, and a corresponding Simpl precondition $\{\!|List\ p\ \mathsf{next}\ Ps|\!\}$ for some universally quantified p and Ps. List Ps has type *ref list*, whereas Ls has type *loc list*, and similarly for p. We first have to relate those logical variables. We instantiate the Simpl specification with $P_a = \{\!|List\ (the\text{-}Ref\ p)\ \mathsf{next}\ (map\ Rep\text{-}loc\ Ls)|\!\}$. P_a depends on the state and we have to exploit the relation between Simpl and C0 states to establish $abs_a\ pn\ P \subseteq P_a$. By unfolding the definition of abs_a we obtain a C0 state s and a Simpl state s_a for which $s_a \in abs_s\ ''Rev''\ s$. Moreover, from the precondition P we obtain the conformance of the state: $TE\vdash s :: HT,empty,GT$, welltypedness of value p: $\lfloor HT\rfloor\vdash_v p :: Ptr\ ''list''$, and the heap list: $List_{C0}\ p\ (heap\ s)\ Ls$. From these assumptions we want to derive $List\ (the\text{-}Ref\ p)\ (next\ (globals\ s_a))\ (map\ Rep\text{-}loc\ Ls)$. We do induction on the list Ls and exploit the definition of the state abstraction abs_s. Predicate $List_{C0}$ guarantees that each next pointer is of the kind $Addr\ l$. The state abstraction provides us with

$$\forall l.\ next\ (globals\ s_a)\ (Rep\text{-}loc\ l) = the\text{-}Ref\ (sel_v\ (the\ (h\ l),\ [''next''])).$$

This is exactly what we need to relate the lists. Since the crucial typing issues are already encoded in $List_{C0}$ we did not have to use the conformance of the initial state.

For the postcondition of the factorial we start with the Simpl assertion and have to transform it to the C0 one:

$$concr\ ''Fac''\ \{\!|\mathsf{Res}_n = fac\ n|\!\} \subseteq \{t.\ lvars\ t\ ''Res'' = \lfloor Prim\ (Unsgnd\ (fac\ n))\rfloor\}.$$

This time the assertions refer to the state, namely to the result variable. By unfolding the definition of *concr*, we again obtain a C0 state t and a corresponding Simpl state $t_a \in abs_s$ *"Fac"* t. From the state abstraction we have:

$$lvars\ t\ \text{"}Res\text{"} \neq None \longrightarrow Res_n\ t_a = the\text{-}Unsgnd_v\ (the\ (lvars\ t\ \text{"}Res\text{"})).$$

This is too weak to derive the desired C0 postcondition. We neither know whether *"Res"* is defined in *lvars t*, nor in case there is a defined value, which type it has. That is why Corollary 8.40 additionally gives us information about conformance and the set of assigned (defined) variables of the final state. Definite assignment guarantees that *lvars t "Res"* is defined and the state conformance ensures that the stored value is indeed an unsigned integer. With this additional information the state abstraction is strong enough and we can derive the desired C0 postcondition.

For the postcondition of the list reversal the situation is similar. Only with definite assignment we know that the result value is defined, and conformance of the state ensures that every value along the pointer chain indeed stores an address. To work around the definedness issue of *"Res"* we could also think of strengthening the state abstraction by removing the premise *lvars t "Res"* $\neq$ *None*. Unfortunately, this breaks the important context switch property $abs_s\ pn\ s \subseteq abs_s\ qn\ (s(\!|lvars := empty|\!))$. Just insert "Fac" for *pn* and *qn*. Then we would have to show that assuming equation $Res_n\ t_a = the\text{-}Unsgnd_v\ (the\ \lfloor v \rfloor)$ for the right hand side, implies the equation $Res_n\ t_a = the\text{-}Unsgnd_v\ (the\ None)$, when the local variables are reset. This is not possible.

An other question is, whether we can get rid of the dependency of the state conformance. The answer is yes, if we write the assertions on the C0 level in a fully "destructive" way. For the factorial this means that we do not specify the postcondition as

$$lvars\ t\ \text{"}Res\text{"} = \lfloor Prim\ (Unsgnd\ (fac\ n)) \rfloor$$

but instead as

$$the\text{-}Unsgnd_v\ (the\ (lvars\ t\ \text{"}Res\text{"})) = fac\ n.$$

This exactly fits to the specification of $Res_n\ t_a$ that we get from the state abstraction and hence we do not need to exploit the conformance of state t to conclude that the destructor form and the constructor form are actually the same.

The conformance assertion in the precondition is imposed by the transfer theorems. So we cannot just remove it. However, we can refine our notion of validity of a Hoare triple. In general we are only interested in specifications of welltyped programs and conforming states. Hence we can incorporate these constraints directly into validity:

◂ Definition 8.57

$$\begin{array}{l}
\Pi,L \models^{T}_{C0} P\ c\ Q \equiv \\
\forall s\ t\ HT\ LT\ A. \\
\quad wf\text{-}prog\ \Pi \land \\
\quad L = dom\ LT \land \\
\quad tnenv\ \Pi \Vdash s :: HT,LT \restriction_A, genv\ \Pi \land \Pi, genv\ \Pi\ {+}{+}\ LT,HT \vdash c\ \surd \land \mathcal{D}\ c\ L\ A \longrightarrow \\
\quad \Pi,L \vdash_{C0} \langle c,\lfloor s \rfloor \rangle \Rightarrow t \longrightarrow s \in P \longrightarrow t \in Some\ \text{'}\ Q
\end{array}$$

This validity notion restricts partial correctness to wellformed programs, a confirming initial state and a welltyped statement that is definitely assigned. Oheimb [84] uses a similar definition. From type safety we can also derive conformance of the final state. With this modified definition of validity, we can keep state conformance under the hood and do not have to mention it in every assertion. Whenever we need conformance we can simply assert it. With this definition we can indeed reduce the precondition of the C0 factorial to $\{s.\ n \leq 12\}$.

How does the idea of "destructive" assertions work for heap data? The predicate $List_{C0}$ is not completely "destructive". It restricts value v do be either *Prim Null* or *Prim* (*Addr l*). The calculation of the next pointer via sel_v (*the* (*h l*), ["*next*"]) however is already "destructive". We can define a destructor *the-Ptr* for the C0 level that works analogously to *the-Ref* for Simpl.

Definition 8.58 ►

$$\begin{array}{lcl@{\qquad}lcl} \textit{the-Ptr} :: \textit{val} \Rightarrow \textit{prim} & & & \textit{the-Ref} :: \textit{val} \Rightarrow \textit{ref} & & \\ \textit{the-Ptr}\ (\textit{Prim Null}) & = & \textit{Null} & \textit{the-Ref}\ (\textit{Prim Null}) & = & \textit{NULL} \\ \textit{the-Ptr}\ (\textit{Prim}\ (\textit{Addr}\ l)) & = & \textit{Addr}\ l & \textit{the-Ref}\ (\textit{Prim}\ (\textit{Addr}\ l)) & = & \textit{Rep-loc}\ l \\ \textit{the-Ptr}\ \text{-} & = & \textit{Null} & \textit{the-Ref}\ \text{-} & = & \textit{NULL} \end{array}$$

The last equations in the definitions are used to make both functions work the same for non pointer values. Now we can modify the list predicate to be fully "destructive":

Definition 8.59 ►

$$\begin{array}{lcl} List_{C0}' :: val \Rightarrow (loc \rightharpoonup val) \Rightarrow loc\ list \Rightarrow bool & & \\ List_{C0}'\ v\ h\ [] & = & \textit{the-Ptr}\ v = Null \\ List_{C0}'\ v\ h\ (l{\cdot}ls) & = & \textit{the-Ptr}\ v = Addr\ l \wedge List_{C0}'\ (sel_v\ (the\ (h\ l),\ [''next'']))\ h\ ls \end{array}$$

With this specification we only need the properties of state abstraction to translate between *List* and $List_{C0}'$. Conformance of the state is no longer necessary, but still we have to do an induction on the list, since *List* and $List_{C0}'$ are different recursive definitions. However, we can take a further step. Instead of introducing a new predicate $List_{C0}'$ we reuse the *List* predicate, but abstract the complete heap:

Definition 8.60 ►

$$\begin{array}{l} next_a :: state \Rightarrow ref \Rightarrow ref \\ next_a\ s\ r \equiv \textit{the-Ref}\ (sel_v\ (the\ (heap\ s\ (\textit{Abs-loc}\ r)),\ [''next''])) \end{array}$$

This projection of the "*next*" component of the heap coincides with the state abstraction:

Lemma 8.48 ► *If* $s_a \in \textit{abs-glob}\ (heap\ s)\ (\textit{free-heap}\ s)\ (gvars\ s)$ and $p \neq NULL$ then
$next_a\ s\ p = next\ (globals\ s_a)\ p$.

For predicate *List* there is a lemma that allows to exchange two heaps if they store the same content for the list elements.

Lemma 8.49 ► *If* $\forall x{\in}set\ ps.\ x \neq NULL \longrightarrow h\ x = g\ x$ then $List\ p\ h\ ps = List\ p\ g\ ps$.

Proof. By induction on list *ps*. □

Together with Lemma 8.48 we can prove that the C0 and Simpl list predicate coincide, without a further induction.

Lemma 8.50 ► *If* $s_a \in \textit{abs-glob}\ (heap\ s)\ (\textit{free-heap}\ s)\ (gvars\ s)$ then
$List\ (\textit{the-Ref}\ p)\ (next_a\ s)\ Ps = List\ (\textit{the-Ref}\ p)\ (next\ (globals\ s_a))\ Ps$.

This approach somehow reflects the canonical interpretation of a Simpl assertion on the C0 level. If we provide lemmas like Lemma 8.49 for all the data-structures we use in a program, the Simpl assertions are canonically lifted to the C0 version and the correspondence proof becomes trivial.

Of course "destructive" specifications also have a drawback. They are weaker as the type constraining variants. It depends on the complexity of the involved values and the properties we are interested in, if welltypedness of the values is crucial for further reasoning. For example, for simple state updates of primitive values we can still conclude that an update of variable *"x"* does not affect variable *"y"*:

$$\textit{the-Unsgnd}_v\ (\textit{the}\ ((\textit{lvars}\ t(''x'' \mapsto v))\ ''y'')) = \textit{the-Unsgnd}_v\ (\textit{the}\ (\textit{lvars}\ t\ ''y'')).$$

This works fine, since the reasoning completely takes place locally inside the destructors:

$$(\textit{lvars}\ t(''x'' \mapsto v))\ ''y'' = \textit{lvars}\ t\ ''y''.$$

However, consider the more complex $\textit{sel}_v$ (*the* (*h l*), [*"next"*]) and an update of the heap at location *l* but in component *"cont"*:

$$\begin{array}{l}\textit{sel}_v\ (\textit{the}\ ((h(l \mapsto \textit{upd}_v\ (\textit{the}\ (h\ l), [''cont''], [], v)))\ l), [''next'']) = \\ \textit{sel}_v\ (\textit{upd}_v\ (\textit{the}\ (h\ l), [''cont''], [], v), [''next''])\end{array}$$

To conclude that the $\textit{upd}_v$ is irrelevant and the right hand side can be further reduced to $\textit{sel}_v$ (*the* (*h l*), [*"next"*]) we need to exploit the type information. If we have a proper conformance constraint for the heap at hand we can get the information from there. We can also add this constraints to predicates like $\textit{List}_{C0}$ which has the advantage that the information is right there where we need it.

Let us take the assertion $\textit{List}_{C0}\ p\ (\textit{heap}\ s)\ \textit{Ls}$ from the precondition of list reversal, where *p* and *Ls* are logical variables. We want to ensure that the type of every value *the* (*h l*), where $l \in \textit{set Ls}$, indeed is a list structure. The type of a location is determined by the heap typing *HT* which itself is just an opaque logical variable. We currently have no information about the type of a location *l*. We can gain it when we know the type of the initial pointer *p*. Given the information that *p* points to a list ($\lfloor HT \rfloor \vdash_v p :: \textit{Ptr}\ ''list''$) and the conformance of the state we can propagate the information through the list. If we want to apply the same lines of reasoning to the postcondition we need to know the type of *"Res"* and that the value of *lvars t "Res"* is defined and indeed a pointer to a list. At first we lack this information since we have not put it to the postcondition. However, Corollary 8.40 allows us to put state conformance and the definedness of variable *"Res"* into the postcondition. Moreover, if we work with the alternative definition of validity we can obtain this information ad-hoc as we need it:

$$\frac{\Pi,L \models^{T}_{C0} P\ c\ Q}{\Pi,L \models^{T}_{C0} P\ c\ \{t.\ \exists HT\ LT.\ \Pi,\textit{genv}\ \Pi\ {+}{+}\ LT,HT \vdash c\ \surd \wedge \textit{dom}\ LT \cap \mathcal{A}\ c \subseteq \textit{dom}\ (\textit{lvars}\ t) \wedge t \in Q\}}$$

This rule allows us to strengthen any postcondition with welltypedness of the statement, conformance of the final state and the guarantee about assigned variables from the definite assignment analyses. The soundness of this rule is given by the type safety Theorems 7.13 and 7.16. Applied to our example we can infer the type of variable *"Res"* since we know the procedure call is welltyped and the procedure definition makes the result type explict. Moreover, the variable has a defined value according to the analysis result of definite assignment. Together with the state conformance we can infer the type of the list elements.

Let me conclude. The "destructive" style of C0 assertions can be obtained canonically from the Simpl assertions. Moreover, if we use the alternative definition of validity it appears that the assertions are strong enough, since we can reconstruct type information in an ad-hoc fashion if necessary. However, dependent of the concrete application it may still be preferable to use stronger assertions that already incorporate the crucial parts of the type information. Then it is unnecessary to exploit state conformance in order to access the information.

8.9 Conclusion

This chapter presented the embedding of C0 into Simpl. This allows to employ the verification environment for Simpl to derive properties of C0 programs. Since this translation is verified the program properties can be transferred back to the C0 level. The translation and the correctness proof illustrate how one can switch from a deep embedding, tailored for meta theory, to a shallow embedding for program verification. The type-soundness results for C0 allow to adopt a simpler model for the verification of individual programs. In particular programming language types can be mapped to HOL types and the split heap model rules out aliasing between different structure fields and pointer types.

This is the first formal verification of the split heap approach. As the language model of Simpl and the programming language C0 are both developed in the same logical framework of Isabelle/HOL the soundness of the embedding is formally verified and machine checked. In contrast, the embedding of C [34] and Java [65] to the Why tool have to be trusted, as the tool is external.

The soundness theorem is modular with respect to the implementation of arithmetic in the program logic. We can either employ the bounded modular arithmetic of C0 or switch to unbounded arithmetic, protected by guards against over- and underflows. Both have their favours and it depends on the application which approach is to prefer. For example, to implement a big-number library or cryptographic primitives it may be convenient to stay within modular arithmetic. Whereas in other applications it is preferable to model and specify the procedures with unbounded arithmetics.

CHAPTER **9**

Conclusion and Outlook

Contents

9.1 Practical Experiences

This section summarises and discusses the practical experiences with the verification environment. The main applications that were so far tackled with the verification environment are the following:

- Case study: Normalisation of Binary Decision Diagrams
- C0 verification in Verisoft
  - C0 Compiler
  - Operating system kernel
  - Email-client
- Memory Management of L4 kernel, with methodology of refinement

9.1.1 Normalisation of Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are a canonical, memory efficient pointer structure to represent Boolean functions, with a wide spread application in computer science. They had a decisive impact on scaling up the technology of model checking to large state spaces and handling practical applications [23]. BDDs were introduced by Bryant [19], and later on refined by Bryant *et al.* [18]. The efficiency of the BDD algorithms stems from the sharing of subgraphs in the BDD.

The basic encoding of the Boolean function in a BDD is a binary decision tree, where each node represents an input variable of the function. An inner node of the BDD contains a Boolean variable over which the function is defined, together with a pointer to the left and right sub-BDD. Given a valuation of the variables the value of the Boolean function encoded in the BDD is obtained by traversing the BDD according to the valuation of the variables. The leaf that is reached holds the value of the function under the given valuation.

The binary decision tree is stored as a directed acyclic graph (DAG) in the BDD. This allows to share common subtrees. Normalisation of a BDD means to remove all redundant nodes and to share all subtrees where possible. A normalised BDD provides an unique representation of the underlying Boolean function. Moreover, the shared representation saves storage space and computation time.

The main challenges for the verification of the normalisation algorithm are:

- sharing of subtrees, and
- an auxiliary data-structure for working on the breadth of the decision tree is introduced.

To formalise the notion of a BDD the abstraction techniques described in Section 4.6 are adapted. In a first step the BDD in the heap is abstracted to a DAG. Then we abstract the DAG to a binary decision tree, and finally the decision tree is interpreted as a Boolean function. The DAG layer is employed to reason about sharing properties. The decision tree layer is used to identify redundant nodes that can be removed, and finally the Boolean function level describes the semantics of the BDD. It has to be preserved by the normalisation algorithm.

Before the main part of the normalisation starts, the BDD is traversed and an additional data structure is built, that links together all the nodes that correspond to the same decision variable. This data-structure is called *level-list*. It is used to process the BDD in a breadth first fashion starting from the leafs up to the root. During the verification of the algorithm we have to keep track of the original BDD, the level-list and the already processed parts that result in the normalised BDD.

The normalisation algorithm is split into 5 procedures and is about 50 lines of Simpl code. The verification of partial correctness of the normalisation algorithm and its auxiliary procedures sums up to about 10 000 lines of Isabelle/Isar formalisation and proofs and is based on a master thesis [87]. The verification work took about 3 person months. Adapting the proofs to total correctness [88] was straightforward and only adds a few lines.

We locate the reasons of the complexity mainly in the data structure, which involves a high degree of data sharing and side effects, which results in quite complex invariants, specifications and proofs. We have to keep track of the original BDD, the level-list and the normalised parts of the BDD.

The hardest part of the proof was to find the invariant for the main loop, that processes the BDD from the leafs to the root in a breadth first fashion. To isolate the loop from the surrounding code the technique described in Section 4.9 was employed. To prove the verification conditions, we used the structured language Isar [114, 116, 79] that allows to focus on and keep track of the various aspects of the proof, so that we can conduct it in a sensible order. Moreover, it turned out that the Isar proofs are quite robust with regard to the iterative adaptation of the invariant resulting from failed proof attempts. The already established lines of reasoning remained stable, while adding new aspects to, or strengthening parts of the invariant. The relatively large size of the proofs is partly explained by the fact that the declarative Isar proofs are in general more verbose than tactic scripts.

The Hoare logic framework and the split heap model appeared to form a suitable verification environment on top of Isabelle/HOL. The abstraction of pointer structures to HOL datatypes allows us to give reasonable specifications. The split heap model addresses parts of the separation problems that occur when specifying procedures on pointer structures. The overhead of describing the parts of the heap that do not change is kept small. The main effort of the work goes into the problem and not into the framework.

9.1.2 C0 Verification in Verisoft

The Verisoft project aims at the pervasive verification of computer systems, comprising hardware, system software, the operating system and user applications. To handle the complexity, the computer system and also the models for verification are organised in layers.

At the bottom there is (i) the hardware layer, on top of it (ii) a machine language (assembler) layer, and on top of it (iii) the programming language layer for C0. Correctness theorems for the components are often simulation theorems between adjacent layers. For example, compiler correctness is a simulation between Layers (ii) and (iii).

Gargano *et al.* [37] introduce an abstract parallel model of computation called *communicating virtual machines* (CVM). It formalises the interaction of concurrent user processes with an operating system kernel. In this model the user processes are virtual machines, which means processors with virtual memory. The so called *abstract kernel* however, is represented as a C machine. This emphasises that the user processes are black-boxes for the operating system kernel. They can attempt to execute any machine code. It is the responsibility of the hardware and primarily of the kernel to ensure that no user process can corrupt the system. The kernel itself is written in C0, augmented with some in-line assembler parts. On the level of the CVM these in-line assembler parts are abstracted to so called *CVM primitives* that alter the state of user processes. For instance, there are CVS primitives to increase and decrease the memory size of an user process or to copy data between processes or I/O devices. Since the kernel is in large parts written in C0 its verification is carried out in the verification environment presented in this thesis. The theorems in Section 8.7 are used to transfer the properties from Simpl to C0. Further a compiler correctness theorem can be employed to transfer the properties to the assembler layer, where they can be combined with the in-line assembler parts. To reduce the reasoning on the assembler layer even more, we lift the effect of the assembler instructions to the C machine. This is the purpose of the CVM primitives. They are specifications for the in-line assembler parts. They modify parts of the system configuration that are usually not visible from a C0 program. However, since the state space in Simpl is polymorphic we can simply extend it with those parts. Since there is no C0 implementation for the CVM primitives they are "implemented" by their specification. For this purpose the *Spec* command is used in Simpl. This allows to reason about the effect of the in-line assembler parts on the abstract Simpl level. The flexibility of Simpl makes these extensions immediately available to the user.

We plan to extend the C0 language with a statement similar to *Spec*. Then the property transfer theorems of Section 8.7 can be adapted to this extended semantics. A correctness proof for the translation (compilation) from C0 with this *Spec* statements to *C0* with in-line assembler guarantees that the assembler parts meet their specification and the whole program has the expected behaviour.

A major effort in Verisoft is to prove the C0 compiler correct [62]. The correctness of the compiler is crucial to ensure soundness of the overall system. As mentioned above, major parts of the reasoning about the operation system kernel takes place on the C0 level. However, in the end this kernel has to be compiled to run on the actual hardware. The compiler correctness is established in two steps. First the *compiling specification* is formalised in Isabelle/HOL and proven correct. The compiling specification is a HOL function that compiles the C0 program to the assembler language. The correctness theorem of the compiling specification is a step by step simulation of the C0 small-step semantics and the semantics of the assembler language. In a second step the compiler is itself implemented in C0. The C0 implementation is proven to implement the compiling specification. This proof is carried out in the verification environment presented in this thesis. The proof mainly is concerned with the different paradigms of the compiling specification and implementation. The specification is basically a functional program in HOL and the implementation is an imperative program. The equivalence of (nested) while loops to recursive functions has to be shown. The syntax tree, the assembler program as well as auxiliary data like type and function tables, are represented as heap structures. The techniques described in Section 4.6 are used to abstract them to the corresponding entities on the specification level.

The C0 implementation of the compiler is over 1 000 lines of C0 code. The formal proof of the compiler implementation comprises about 900 lemmas and a total of 20 000 proof lines. It took about 1 person year. The proof is for partial correctness and the guards where omitted. From the experience in the BDD case study, we expect that the adaptation to total correctness is straightforward. As described in Section 5, we plan to use a software model checker to discharge the guards.

On top of the operating system an E-mail client is implemented as a demo application. The operating system supports the SMTP protocol and the E-mail client encrypts the messages. The E-mail client is implemented and specified [12] and the verification is ongoing work. Unfortunately there are no further publications available up to now.

Another subproject of Verisoft applies system verification to embedded devices. In this domain an extension of C called DPCE[1] is used that introduces data-parallel instructions. Since Simpl is not fixed to a particular programming language the C0 embedding was extended with these data parallel instructions.

9.1.3 Memory Management of L4 Kernel, with Refinement

The project *L4.verified*[2] is also concerned with the verification of operating systems. In this case the L4 micro kernel [63]. They employ the methodology of data refinement [28] to structure their verification. Tuch and Klein [109] have verified the virtual memory subsystem of the L4 kernel with this approach. They start with an

[1]See the DPCE home page for more information: `http://www.crescentbaysoftware.com/dpce/`
[2]`http://www.cse.unsw.edu.au/~formalmethods/projects/l4.verified/`

abstract view on the virtual memory system and prove the crucial properties on this level. The system is defined semantically as an abstract data type that consists of a set of initial states and a set of operations that specify possible state transitions of the system. Then they refine the model until a C implementation is reached. The meta theory of data refinement ensures that the properties for the abstract system are preserved by the refinement step, as long as each of the operations is implemented correctly. As the C level is reached the Hoare logic is used to prove the correctness of the individual operations. The soundness Theorems 3.8 and 3.15 for the Hoare logic are used to interpret the Hoare triple as a state transition specification so that the result can be used in the refinement framework. Moreover, the *Spec* command of Simpl can be used to define refinement between Simpl programs. On the higher level the operation is just specified, without implementation. As the refinement proceeds the specification is implemented by a concrete Simpl statement that meets the specification.

As the L4 kernel implementation also involves pointer arithmetic Tuch and Klein [110] have developed an alternative heap model for Simpl that is capable to deal with low-level manipulations like pointer arithmetic in untyped memory, but still offers a neat, abstract and typed view of memory where possible.

9.2 State Space Representation Revisited

For the reasons discussed in Section 2.4.1 we decided to use records as the state space representation. The main benefit of this approach is that primitive types of the programming language coincide with types in HOL and hence the specifications and verification conditions are quite natural. The drawback of the records is that the type of the state space is not uniform, since it depends on the variables appearing in the program. Moreover, the field names of records are no first class objects in the logic and hence we cannot do much meta-level reasoning about records inside the logic. We managed to express all crucial properties without this quantification. However, the abstraction from C0 states to Simpl states, for instance, cannot be formalised generically in HOL for this reason. Hence we postponed this translation to the point where the C0 program and the corresponding state record can be fixed. Another issue of records is scalability. Records are a heavyweight feature of HOL. For each record field a selector and update function has to be defined together with means to simplify selections and updates by rewriting. Internally, records are implemented as nested pairs. To prove that an update of field x does not affect a selection of field y basically requires to split the record to its components. The more fields the record has the more costly this operation becomes. In contrast, if we represent the state as a function from names to values, we only have to compare the names regardless of how many names are present in the program.

The simplicity of specifications and the verification conditions is a crucial prerequisite for the usability of the tool that we do not want to sacrifice. So the question is if we can provide a more lightweight representation of the state space but still can keep the simplicity of verification conditions. According to the discussion in Sections 2.4.1, 8.8 and 2.4.9.1 the most promising candidate is the function from names to values, together with the fully "destructive" scheme for specifications and state updates. For the split heap model we can use the function from field-name and reference to value. Of course, as detailed in Section 2.4.1, this leads to various value constructors and destructors that have to be inserted into the terms. However, with some additional implementation effort it might be possible to hide this clutter from the user. The constructors and destructors could be inserted by automatic syntax translations in Isabelle, in order to support simple specifications. However, this mere syntactic sugar does not prevent the destructors to show up in the verification condition. We end up with proof obligations like:

$$\bigwedge s.\ \textit{the-Intg}_v\ (\textit{lvars}\ s\ ''n'') < m,$$

instead of

$$\bigwedge n.\ n < m,$$

what we obtain right now. Again, with additional implementation effort, one can clean up the verification condition by generalising it. We can abstract every occurrence of $\textit{the-Intg}_v$ $(\textit{lvars}\ s\ ''n'')$ in the verification condition to an universally quantified variable n. This results in exactly the same proof obligation as we obtain right now. Nested quantification over the state, as introduced by the verification condition for procedure calls, complicates the generalisation. Moreover, with the representation of the split heap model as a single function with two parameters it becomes necessary to distribute destructors over heap updates. However, it still seems possible to automate such a generalisation.

Since the state space of Simpl is polymorphic, we do not have to change anything about the basic Hoare logic, or the meta-theory of Simpl. Even the embedding of C0 to Simpl remains valid, since it is open to different state space representations in the Simpl layer. It may even be possible to discharge the program specific requirements once and for all, since we can formalise the state abstraction, lookup and update function generically. Hence it seems to be a worthwhile project for the future to investigate this alternative state space representation.

9.3 Contributions

I think the main contribution of this thesis is to provide a practically useful verification environment for imperative programs, without making any concessions on the meta theory of the tool. Having the meta theory available in the same uniform framework of HOL, makes it possible to embed realistic programming languages and to extend the calculus without introducing any soundness risks. Examples for those extension are the integration of program analysis or the tool support for composing verified libraries. As is being demonstrated by the aforementioned large-scale verification projects, this framework has finally made practical, concrete program verification feasible.

In this thesis I have introduced a general language model for sequential imperative programs, its operational semantics and a Hoare logic for partial and total correctness. The language model is expressive enough to cover all common language features, like mutually recursive procedures, abrupt termination and exceptions, runtime faults, local and global variables, pointers and heap, expressions with side effects, pointers to procedures, partial application and closures, dynamic method invocation and also unbounded nondeterminism. Despite its expressive power, the language model and its meta theory is still neat and clean. Soundness and completeness of the Hoare logics for both partial and total correctness have been proven. Especially the completeness proof for total correctness of such an expressive language goes beyond the related work in the area of formalised program calculi.

Furthermore, I have clarified the handling of auxiliary variables and the consequence rule. The auxiliary variables are now completely "auxiliary" and do not appear in the core Hoare calculus anymore. This avoids type restrictions on the auxiliary variables and allows any number of auxiliary variables in the specifications. This gives the user the freedom to write natural specifications, which is a crucial ingredient for the practical usability of the tool.

All specifications and proofs have been carried out in the interactive theorem prover Isabelle/HOL. All theorems in this thesis are machine checked and generated from the Isabelle sources.

The following table gives an overview of the size of the formalisation.

Formalisation	Lines of Isabelle code	Pages of proof document
Simpl	27 400	524
C0[3]	5 800	220
Embedding C0 into Simpl	18 800	400

The Hoare logic is mechanised and integrated into Isabelle/HOL. An automatic verification condition generator is implemented as Isabelle tactic and Isabelle's locales are employed to organise procedure specifications. This makes the comprehensive infrastructure of Isabelle/HOL accessible for the verification of imperative programs. Moreover, I have developed a framework to build and reuse generic verified libraries. The following table lists the sizes of the implementation.

Implementation	Lines of ML code
Hoare module	2 800
Syntax translations for Simpl	1 300

The BDD case study and the experiences in the large-scale verification projects Verisoft and L4.verified document that the verification environment is practically useful, effective, extensible and flexible.

The Hoare logic features extended means to handle guards. This provides a clean interface to automatic program verification tools like software model checkers or program analysis. The results of the program analysis can be introduced to the Hoare logic in form of discharged guards in the program text. These guards are treated as granted for the rest of the verification. This scheme can be used to delegate the handling of runtime faults like arithmetic overflows or dereferencing null pointers to a software model checker. Hence the interactive proof can focus on functional correctness. Moreover, the same scheme can be employed to introduce general properties that a program analysis has inferred to the verification environment. They can be used to support the interactive proof.

The language model is not restricted to a particular programming language. Instead it can be used to embed a programming language in order to do program verification. I have demonstrated this by embedding C0 into Simpl and proving the soundness of this translation. In order to verify a C0 program it is first translated to

[3]Large parts of the C0 formalisation go back to the work of Martin Strecker for the Verisoft project.

Simpl. Then the verification environment can be used to proof program properties. Finally, the soundness of the translation allows to transfer the properties to C0 again. The translation of C0 to Simpl shows how the type-safety of C0 can be exploited in order to obtain a simpler model for the verification of individual programs. Primitive programming language types are directly mapped to HOL types. Hence we do not have to care about typing issues during program verification, since it is already covered by the type inference of Isabelle/HOL. Moreover, the monolithic heap in C0 is translated to a split heap model in Simpl, which excludes aliasing between structure fields. The soundness theorem justifies this translation. From a methodological point of view the soundness proof for the translation from C0 to Simpl shows how a deep embedding of expressions, tailored for meta-theory, can be transferred to a shallow embedding for the purpose of program verification.

9.4 Further work

The work in this thesis can serve as basis for further investigations in various directions:

Usability In realistic applications, the verification condition generator generates quite sizeable proof obligations, that follow the control flow of the program. In the BDD case study we manually decomposed these proof obligations to a structured Isar proof. The skeleton of such an Isar proof could be directly generated in addition to the proof obligation.

Embeddings I have presented the embedding of C0 into Simpl. It would be interesting to embed further programming languages, in particular object oriented languages like Java. Dynamic method invocation can be mapped to the dynamic command in Simpl.

Extensions Simpl is restricted to sequential programs. The natural next step is to extend Simpl with parallelism. On the one hand one can implement shared state parallelism, as formalised by Prensa [98] for a while language. On the other hand an integration with a process calculus like CSP [49] can be investigated.

Assertions The Hoare logic allows arbitrary HOL predicates as assertions. In recent years separation logic [101] was proposed to reason about heap data. Right now there is no tool support for separation logic in interactive theorem provers. It seems desirable to investigate if and how separation logic can be integrated into the verification environment.

Refinemet The integration of Simpl into the refinement framework that was initiated by Tuch and Klein [109] can be extended. The challenge is to keep the overhead of the refinement meta theory at a minimum, so that it does not become a burden for practical applications.

State As already elaborated in Section 9.2 it seems worthwhile to experiment with alternative state space representations.

Wir stehen selbst betrübt und sehn betroffen,
Den Vorhang zu und alle Fragen offen.
— Bertolt Brecht

APPENDIX **A**

Theorems about Correlation of the Simpl Big- and Small-Step Semantics

In this chapter, we investigate the various correlations between the Simpl big- and small-step semantics and the associated notions of termination and infinite computation.

Every big-step execution can be embedded into a small-step computation.

If $\Gamma\vdash \langle c,s\rangle \Rightarrow t$ *then* $\forall cs\ css.\ \Gamma\vdash \langle c{\cdot}cs,\ css,\ s\rangle \rightarrow^{*} \langle cs,\ css,\ t\rangle$. ◂ Lemma A.1

Proof. By induction on the big step execution. □

By instantiating *cs* and *css* with the empty list we get the simulation of the big-step execution in the small-step semantics.

If $\Gamma\vdash \langle c,s\rangle \Rightarrow t$ *then* $\Gamma\vdash \langle [c],\ [],\ s\rangle \rightarrow^{*} \langle [],\ [],\ t\rangle$. ◂ Lemma A.2 *Small-step simulates big-step*

A small-step configuration has richer structure than a big-step configuration. The first component of a small-step configuration is a statement list instead of a single statement in the big-step semantics. Moreover, the small-step configuration has a continuation stack which is completely missing in the big-step semantics. To prove the simulation of a terminating small-step execution by the big-step semantics we first extend the big-step semantics to statement lists and a continuation stack.

The extended operational big-step semantics: $\Gamma\vdash \langle cs,css,s\rangle \Rightarrow t$, *is defined inductively by the rules in Figure A.1. Execution of the initial configuration* $\langle cs,\ css,\ s\rangle$ *in procedure environment* Γ *leads to the final state* t*. Where:* ◂ Definition A.1 *Extended big-step semantics for Simpl*

Γ :: $'p \rightharpoonup ('s,\ 'p,\ 'f)\ com$	cs :: $('s,\ 'p,\ 'f)\ com\ list$
s, t :: $('s,\ 'f)\ xstate$	css :: $('s,\ 'p,\ 'f)\ continuation\ list$

The statement sequence *cs* is consecutively executed by the ordinary big-step semantics. If it is completely processed the continuation stack is considered analogously to the small-step semantics.

A terminating small-step computation can be simulated by the extended big-step semantics.

If $\Gamma\vdash \langle cs,\ css,\ s\rangle \rightarrow^{*} \langle [],\ [],\ t\rangle$ *then* $\Gamma\vdash \langle cs,css,s\rangle \Rightarrow t$. ◂ Lemma A.3

$$\frac{}{\Gamma\vdash \langle [],[],s\rangle \Rightarrow s}\ (\textsc{Nil}) \qquad \frac{\Gamma\vdash \langle c,s\rangle \Rightarrow s' \qquad \Gamma\vdash \langle cs,css,s'\rangle \Rightarrow t}{\Gamma\vdash \langle c\cdot cs,css,s\rangle \Rightarrow t}\ (\textsc{Cons})$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$\frac{\Gamma\vdash \langle nrms,css,Normal\ s\rangle \Rightarrow t}{\Gamma\vdash \langle [],(nrms,\ abrs)\cdot css,Normal\ s\rangle \Rightarrow t}\ (\textsc{ExitBlockNormal})$$

$$\frac{\Gamma\vdash \langle abrs,css,Normal\ s\rangle \Rightarrow t}{\Gamma\vdash \langle [],(nrms,\ abrs)\cdot css,Abrupt\ s\rangle \Rightarrow t}\ (\textsc{ExitBlockAbrupt})$$

$$\frac{\Gamma\vdash \langle nrms,css,Fault\ f\rangle \Rightarrow t}{\Gamma\vdash \langle [],(nrms,\ abrs)\cdot css,Fault\ f\rangle \Rightarrow t}\ (\textsc{ExitBlockFault})$$

$$\frac{\Gamma\vdash \langle nrms,css,Stuck\rangle \Rightarrow t}{\Gamma\vdash \langle [],(nrms,\ abrs)\cdot css,Stuck\rangle \Rightarrow t}\ (\textsc{ExitBlockStuck})$$

Figure A.1: Extended big-step semantics for Simpl

Proof. By reflexive transitive closure induction.

Case reflexivity: We have to show $\Gamma\vdash \langle [],[],t\rangle \Rightarrow t$, which is covered by the Nil Rule.

Case transitivity: As induction hypothesis we have an initial step of the computation

$$\Gamma\vdash \langle cs,\ css,\ s\rangle \rightarrow \langle cs',\ css',\ s'\rangle, \qquad (*)$$

that we can finish by a big-step execution

$$\Gamma\vdash \langle cs',css',s'\rangle \Rightarrow t. \qquad (**)$$

We have to show that we can also start the big-step execution in the initial configuration:

$$\Gamma\vdash \langle cs,css,s\rangle \Rightarrow t.$$

This is proven by exhaustive case distinction on the initial step (∗) according to the single step relation. □

By specialising the previous lemma to a singe statement we get the simulation of a terminating small-step execution by the ordinary big-step semantics.

Lemma A.4 ► *Big-step simulates terminating small-step*

If $\Gamma\vdash \langle [c],\ [],\ s\rangle \rightarrow^* \langle [],\ [],\ t\rangle$ *then* $\Gamma\vdash \langle c,s\rangle \Rightarrow t$.

Putting Lemma A.2 and Lemma A.4 together we arrive at the equivalence of a terminating small-step computation and the big-step execution.

Theorem A.5 ► *Terminating small-step iff big-step*

$$\Gamma\vdash \langle [c],\ [],\ s\rangle \rightarrow^* \langle [],\ [],\ t\rangle = \Gamma\vdash \langle c,s\rangle \Rightarrow t$$

This theorem closely relates the big-step and the small-step semantics. For the rest of this chapter we turn our attention to the two characterisations of guaranteed termination: The inductively defined judgement $\Gamma\vdash c \downarrow s$ and the absence of infinite computations $\neg\ \Gamma\vdash \langle [c],[],s\rangle \rightarrow \ldots(\infty)$. The inductive variant is nice to use, since it comes along with an induction principle for proving properties about terminating programs. Moreover, the decomposition for compound statements like *Seq* c_1 c_2 is directly built into the rules. We immediately get that c_1 terminates in s and that for every intermediate state that is reachable by executing c_1 in s also c_2 terminates.

However, the alternative characterisation of termination, as absence of infinite computations, also has its favours. First of all it is more intuitive than the termination judgement, since it directly encodes the idea of termination as "no infinite computation". Moreover, the mere fact that we used this characterisation in order to prove the completeness of the Hoare logic for total correctness (cf. Theorem 3.22) makes it indispensable for our argumentation. Viewed that way the inductive termination judgement is questionable, since it was not strong enough for our argumentation. However, as the following equivalence argumentation shows, it is much harder to work with the absence of infinite computations. From a proof engineering point of view the equivalence proof provides us with a separation of concerns. We can use the more convenient inductive termination judgement whenever possible, and only have to show once that the absence of infinite computations shares the same properties.

We start with the direction from $\Gamma\vdash c \downarrow s$ to $\neg\, \Gamma\vdash \langle [c],[],s\rangle \rightarrow \ldots(\infty)$. The basic idea is to do induction on the termination judgement and contradict the assumption that there is an infinite computation. The main lemma that we need to make use of the induction hypotheses for compound statements is the following:

◂ Proposition A.6

If $\Gamma\vdash \langle c{\cdot}cs,css,s\rangle \rightarrow \ldots(\infty)$ *then*
$\Gamma\vdash \langle [c],[],s\rangle \rightarrow \ldots(\infty) \vee (\exists t.\ \Gamma\vdash \langle c,s\rangle \Rightarrow t \wedge \Gamma\vdash \langle cs,css,t\rangle \rightarrow \ldots(\infty))$.

It allows to do a case distinction on infinite computations. Either the execution of the head statement c already leads to an infinite computation, or there is an intermediate state t such that the execution of c terminates in t, and the rest computation is infinitary. Consider the case of sequential composition, for example. Given $\Gamma\vdash$*Seq* c_1 $c_2 \downarrow$ *Normal* s and the corresponding induction hypothesis for c_1 and c_2:

- $\neg\, \Gamma\vdash \langle [c_1],[],\mathit{Normal}\ s\rangle \rightarrow \ldots(\infty)$ (*)
- $\forall t.\ \Gamma\vdash \langle c_1,\mathit{Normal}\ s\rangle \Rightarrow t \longrightarrow \neg\, \Gamma\vdash \langle [c_2],[],t\rangle \rightarrow \ldots(\infty)$. (**)

We assume that there is an infinite computation $\Gamma\vdash \langle [\mathit{Seq}\ c_1\ c_2],[],\mathit{Normal}\ s\rangle \rightarrow \ldots(\infty)$. From the small-step semantics we know that the initial step of the computation is $\Gamma\vdash \langle [\mathit{Seq}\ c_1\ c_2], [], \mathit{Normal}\ s\rangle \rightarrow \langle [c_1, c_2], [], \mathit{Normal}\ s\rangle$. Hence we get an infinite computation starting from the second configuration: $\Gamma\vdash \langle [c_1, c_2],[],\mathit{Normal}\ s\rangle \rightarrow \ldots(\infty)$. With the case distinction proposition A.6 and the induction hypothesis (*) and (**) we get a contradiction.

To prove proposition A.6 we analyse the sequence of configurations of an infinite computation. Consider a computation that starts in configuration $\langle c{\cdot}cs,\ css,\ s\rangle$. As long as execution of the head statement c is not yet finished, the components cs and css are always part of the subsequent configurations. If c is finished the configuration has the form $\langle cs,\ css,\ t\rangle$. To characterise the intermediate configurations is more involved. We relate the configuration sequence to one that is started with the head statement only: $\langle [c],\ [],\ s\rangle$. The configuration sequence obtained from this initial configuration describes the *pure* computation of c. As naming convention we describe parts of these pure configurations with a prefix p, like $\langle pcs,\ pcss,\ t\rangle$. The original computation is referred to as *compound* computation. We can also think of pcs and $pcss$ as the *progress* that is made compared to the initial configuration. It describes the delta between the current configuration and the initial one. As long as no new blocks are entered the pure configuration has the form $\langle pcs,\ [],\ s'\rangle$. Hence the compound configuration has the form $\langle pcs\ @\ cs,\ css,\ s'\rangle$. Let us now consider a general intermediate configuration $\langle pcs,\ pcss,\ s'\rangle$ of the pure computation. When a block is entered, a continuation pair is pushed to the continuation stack. When a block is exited the top of the continuation stack is popped. So the initial continuation stack *css* of the compound computation always remains on the bottom of the stack, as long as *c* is calculated. However, the continuation stack is not just of the form *pcss* @ *css*. The initial statements *cs* have to be appended to the tails of the pure continuation stack *pcss*. Let *pcss* be of the form $pcss'\ @\ [(pcs\text{-}normal,\ pcs\text{-}abrupt)]$. Then the continuation stack of the compound computation is $pcss'\ @\ [(pcs\text{-}normal\ @\ cs,\ pcs\text{-}abrupt\ @\ cs)]\ @\ css$. Altogether the corresponding compound configuration to a pure configuration $\langle pcs,\ pcss,\ s'\rangle$ has either of the following forms:

- $pcss = []$: $\langle pcs\ @\ cs,\ css,\ s'\rangle$
- $pcss \neq []$: $\langle pcs,\ butlast\ pcss\ @\ [\langle fst\ (last\ pcss)\ @\ cs,\ snd\ (last\ pcss)\ @\ cs\rangle]\ @\ css,\ s'\rangle$.

Note that in the second case the statement list *pcs* coincides for the pure and the compound computation, because the initial *cs* has moved to the continuation stack.

This relation between configurations of the pure and compound computation shows up in the following lemmas. Before going into detail with these quite technical lemmas, a few words on the outline of the argumentation. The goal is to prove the case distinction proposition A.6 for an infinite computation, started in an initial configuration $\langle c{\cdot}cs,\ css,\ s\rangle$. While the execution of statement *c* is not yet finished we can relate the compound configurations to the pure configurations $\langle pcs,\ pcss,\ s'\rangle$. If we never reach a configuration $\langle cs,\ css,\ t\rangle$ in the compound computation, this means that we never reach a configuration where $pcs = []$ and $pcss = []$ in the pure computation. Thus the computation of *c* is infinite: $\Gamma\vdash \langle [c],[],s\rangle \rightarrow \ldots(\infty)$. If we arrive in an intermediate configuration of the form $\langle cs,css,\text{-}\rangle$, where the statement *c* is completely executed, then we also arrive in such a configuration for the first time:

$$\Gamma\vdash \langle c{\cdot}cs,\ css,\ s\rangle \rightarrow^* \langle cs,\ css,\ t\rangle.$$

Since it is the first time we arrive in a configuration of the form $\langle cs,css,\text{-}\rangle$, we also get a pure computation:

$$\Gamma\vdash \langle [c],\ css,\ s\rangle \rightarrow^* \langle [],\ [],\ t\rangle.$$

According to Lemma A.4 this corresponds to a big-step execution:

$$\Gamma\vdash \langle c,s\rangle \Rightarrow t.$$

Moreover, since we know that the computation is infinite the remaining computation must be infinite: $\Gamma\vdash \langle cs,css,t\rangle \rightarrow \ldots(\infty)$.

As outlined above the heart of the argumentation is the case distinction whether an intermediate configuration of the form $\langle cs,css,\text{-}\rangle$ is reachable or not. The main proof technique is induction on the first *k* steps of the computation. According to the definition of an infinite computation we talk about a function *f* that enumerates the configurations of the infinite computation. To get hold of the statement list, the continuation stack and the state of a configuration we define the selectors *CS*, *CSS* and *S*:

$$\begin{array}{lcl} CS\ \langle cs,\ css,\ s\rangle & = & cs \\ CSS\ \langle cs,\ css,\ s\rangle & = & css \\ S\ \langle cs,\ css,\ s\rangle & = & s \end{array}$$ ◂ Definition A.2

The following lemma describes the effect of a single computation step for a non-empty statement list:

◂ Lemma A.7

If $\Gamma\vdash \langle c\cdot cs,\ css,\ s\rangle \rightarrow \langle cs',\ css',\ t\rangle$ *then*
$\exists pcss.$
$css' = pcss\ @\ css \wedge$
if $pcss = []$ *then* $\exists ps.\ cs' = ps\ @\ cs$
else $\exists pcs\text{-}normal\ pcs\text{-}abrupt.\ pcss = [(pcs\text{-}normal\ @\ cs,\ pcs\text{-}abrupt\ @\ cs)]$.

Proof. By induction on the single step execution. □

To exit a block the continuation stack is popped according to the current state. The initial *css* suffix is not affected.

◂ Lemma A.8

If $\Gamma\vdash \langle [],\ (pcs\text{-}normal,\ pcs\text{-}abrupt)\cdot pcss\ @\ css,\ s\rangle \rightarrow \langle cs',\ pcss\ @\ css,\ t\rangle$ *then*
case s *of* $Abrupt\ s' \Rightarrow cs' = pcs\text{-}abrupt \wedge t = Normal\ s'$
$|\ \text{-} \Rightarrow cs' = pcs\text{-}normal \wedge t = s$.

Proof. By induction on the single step execution. □

The next lemma lifts the previous two lemmas for a single step of computation to multiple steps:

◂ Lemma A.9

Given an infinite computation $\forall i.\ \Gamma\vdash f\ i \rightarrow f\ (i+1)$, *started in the initial configuration* $f\ 0 = \langle c\cdot cs,\ css,\ s\rangle$, *if a configuration of shape* $\langle cs, css, \text{-}\rangle$ *is not yet reached:*

$$\forall i{<}k.\ \neg\ (CS\ (f\ i) = cs \wedge CSS\ (f\ i) = css),$$

then we can identify progress in the configurations with respect to the initial configuration as follows:

$\forall i{\le}k.\ \exists pcs\ pcss.$
  if $pcss = []$ *then* $CSS\ (f\ i) = css \wedge CS\ (f\ i) = pcs\ @\ cs$
  else $CS\ (f\ i) = pcs \wedge$
    $CSS\ (f\ i) = butlast\ pcss\ @\ [(fst\ (last\ pcss)\ @\ cs,\ snd\ (last\ pcss)\ @\ cs)]\ @\ css.$

Proof. By induction on k. **Case** 0 is trivial. **Case** $k+1$: The configurations $i \le k$ are covered by the induction hypothesis. By case analysis on the configuration of step k, we construct the new progress for step $k+1$ from the progress obtained from the induction hypothesis for step k. Lemmas A.7 and A.8 already capture the essential behaviour of the small step semantics, so we do not need to argue on the different statements here. □

The next lemmas are used to extract the embedded pure computation from a compound computation. We start with lemmas for a single step of computation. A suffix of the continuation stack can be dropped:

◂ Lemma A.10

If $\Gamma\vdash \langle cs,\ pcss\ @\ css,\ s\rangle \rightarrow \langle cs',\ pcss'\ @\ css,\ t\rangle$ *then* $\Gamma\vdash \langle cs,\ pcss,\ s\rangle \rightarrow \langle cs',\ pcss',\ t\rangle$.

Proof. By induction on the single step execution. □

So together with Lemma 3.19 we can add and drop the suffix of the continuation stack.

For the statement list we can only drop a suffix if the continuation stack stays the same.

Lemma A.11 ► *If* $\Gamma\vdash \langle c\cdot cs \mathbin{@} xs,\ css,\ s\rangle \rightarrow \langle cs' \mathbin{@} xs,\ css,\ t\rangle$ *then* $\Gamma\vdash \langle c\cdot cs,\ css,\ s\rangle \rightarrow \langle cs',\ css,\ t\rangle$.

Proof. By induction on the single step execution. □

If a new block is entered the first time the suffix *cs* of the compound computation moves to the continuation stack. This suffix can be removed for the pure computation:

Lemma A.12 ► *If* $\Gamma\vdash \langle p\cdot pcs \mathbin{@} cs,\ css,\ s\rangle \rightarrow \langle cs',\ (pcs\text{-}normal \mathbin{@} cs,\ pcs\text{-}abrupt \mathbin{@} cs)\cdot css,\ t\rangle$ *then* $\Gamma\vdash \langle p\cdot pcs,\ css,\ s\rangle \rightarrow \langle cs',\ (pcs\text{-}normal,\ pcs\text{-}abrupt)\cdot css,\ t\rangle$.

Proof. By induction on the single step execution. □

These lemmas for a single step of the computation are used in the induction step of the following lemma, which allows to construct the pure computation from the compound computation. The induction is on *k*.

Lemma A.13 ► *Given an infinite computation* $\forall i.\ \Gamma\vdash f\ i \rightarrow f\ (i+1)$, *started in the initial configuration* $f\ 0 = \langle c\cdot cs,\ css,\ s\rangle$, *if a configuration of shape* $\langle cs, css, \text{-}\rangle$ *is not yet reached:*

$$\forall i{<}k.\ \neg\ (CS\ (f\ i) = cs \wedge CSS\ (f\ i) = css),$$

and the configurations so far can be split up into the parts describing the progress and the parts from the initial configuration:

$$\begin{array}{l}\forall i{\le}k.\ \textit{if}\ pcss\ i = []\ \textit{then}\ CSS\ (f\ i) = css \wedge CS\ (f\ i) = pcs\ i \mathbin{@} cs\\ \qquad \textit{else}\ CS\ (f\ i) = pcs\ i\ \wedge\\ \qquad\qquad CSS\ (f\ i) =\\ \qquad\qquad butlast\ (pcss\ i) \mathbin{@} [(fst\ (last\ (pcss\ i)) \mathbin{@} cs,\ snd\ (last\ (pcss\ i)) \mathbin{@} cs)] \mathbin{@} css,\end{array}$$

then the corresponding pure computation can be build from the progress parts of the configurations:

$$\forall i{<}k.\ \Gamma\vdash \langle pcs\ i,\ pcss\ i,\ S\ (f\ i)\rangle \rightarrow \langle pcs\ (i+1),\ pcss\ (i+1),\ S\ (f\ (i+1))\rangle.$$

Proof. By induction on *k*.

Case *0* is trivial.

Case $k+1$. For $i < k$ the embedded pure computation is provided by the induction hypothesis. For $i = k$ we construct the last step from i to $i+1$ by case distinction on the configuration of step i and Lemmas A.10, A.11, A.12. As these lemmas already capture the possible changes in the shape of a configuration by one step of execution, we do not need to do a case analysis on the head statement. □

Now we prove the case distinction lemma for infinite computations (proposition A.6).

Lemma A.14 ► *If* $\Gamma\vdash \langle c\cdot cs, css, s\rangle \rightarrow \ldots(\infty)$ *then* $\Gamma\vdash \langle [c], [], s\rangle \rightarrow \ldots(\infty) \vee (\exists t.\ \Gamma\vdash \langle c, s\rangle \Rightarrow t \wedge \Gamma\vdash \langle cs, css, t\rangle \rightarrow \ldots(\infty))$.

Proof. We have an infinite computation $\forall i.\ \Gamma\vdash f\ i \rightarrow f\ (i + 1)$ starting in the initial configuration $f\ 0 = \langle c\cdot cs,\ css,\ s\rangle$. We do case distinction whether a configuration of the shape $\langle cs,css,\text{-}\rangle$ is reachable, which means that the first statement c finished its computation.

Case $\exists i.\ CS\ (f\ i) = cs \wedge CSS\ (f\ i) = css$:
Let k be the least number for which we have

$$CS\ (f\ k) = cs \text{ and } CSS\ (f\ k) = css. \qquad (*)$$

We use Lemma A.9 to identify the progress in the configurations up to step k:

$$\begin{array}{l}\forall i{\le}k.\ \exists pcs\ pcss.\\ \quad \textsf{if}\ pcss = []\ \textsf{then}\ CSS\ (f\ i) = css \wedge CS\ (f\ i) = pcs\ @\ cs\\ \quad \textsf{else}\ CS\ (f\ i) = pcs \wedge\\ \qquad CSS\ (f\ i) = butlast\ pcss\ @\ [(fst\ (last\ pcss)\ @\ cs,\ snd\ (last\ pcss)\ @\ cs)]\ @\ css.\end{array}$$

Via the axiom of choice we obtain enumeration functions pcs and $pcss$ for this progress:

$$\begin{array}{l}\forall i{\le}k.\ \textsf{if}\ pcss\ i = []\ \textsf{then}\ CSS\ (f\ i) = css \wedge CS\ (f\ i) = pcs\ i\ @\ cs\\ \quad \textsf{else}\ CS\ (f\ i) = pcs\ i \wedge\\ \qquad CSS\ (f\ i) =\\ \qquad butlast\ (pcss\ i)\ @\ [(fst\ (last\ (pcss\ i))\ @\ cs,\ snd\ (last\ (pcss\ i))\ @\ cs)]\ @\ css.\end{array} \qquad (**)$$

Now we employ Lemma A.13 to extract the first k steps of the embedded pure computation:

$$\forall i{<}k.\ \Gamma\vdash \langle pcs\ i,\ pcss\ i,\ S\ (f\ i)\rangle \rightarrow \langle pcs\ (i + 1),\ pcss\ (i + 1),\ S\ (f\ (i + 1))\rangle. \qquad (***)$$

Using $(*)$ and $(**)$ we can simplify the initial and final state of the pure computation:

$$\begin{array}{rcl}\langle pcs\ 0,\ pcss\ 0,\ S\ (f\ 0)\rangle &=& \langle [c],\ [],\ s\rangle\\ \langle pcs\ k,\ pcss\ k,\ S\ (f\ k)\rangle &=& \langle [],\ [],\ S\ (f\ k)\rangle.\end{array}$$

With $(***)$ we obtain $\Gamma\vdash \langle [c],\ [],\ s\rangle \rightarrow^* \langle [],\ [],\ S\ (f\ k)\rangle$ by induction on k. According to Lemma A.4 this corresponds to the big-step execution $\Gamma\vdash \langle c,s\rangle \Rightarrow S\ (f\ k)$. Shifting the enumeration function f for k steps yields an infinite computation starting in configuration k: $\Gamma\vdash \langle cs,css,S\ (f\ k)\rangle \rightarrow \ldots(\infty)$. Thus we have derived the right alternative of the thesis.

Case $\forall i.\ \neg\ (CS\ (f\ i) = cs \wedge CSS\ (f\ i) = css)$: With Lemma A.9 we identify the progress in the infinite configurations:

$$\begin{array}{l}\forall i.\ \exists pcs\ pcss.\\ \quad \textsf{if}\ pcss = []\ \textsf{then}\ CSS\ (f\ i) = css \wedge CS\ (f\ i) = pcs\ @\ cs\\ \quad \textsf{else}\ CS\ (f\ i) = pcs \wedge\\ \qquad CSS\ (f\ i) = butlast\ pcss\ @\ [(fst\ (last\ pcss)\ @\ cs,\ snd\ (last\ pcss)\ @\ cs)]\ @\ css.\end{array}$$

Via the axiom of choice we obtain enumeration functions pcs and $pcss$ for this progress:

$$\begin{array}{l}\forall i.\ \textsf{if}\ pcss\ i = []\ \textsf{then}\ CSS\ (f\ i) = css \wedge CS\ (f\ i) = pcs\ i\ @\ cs\\ \quad \textsf{else}\ CS\ (f\ i) = pcs\ i \wedge\\ \qquad CSS\ (f\ i) =\\ \qquad butlast\ (pcss\ i)\ @\ [(fst\ (last\ (pcss\ i))\ @\ cs,\ snd\ (last\ (pcss\ i))\ @\ cs)]\ @\ css.\end{array} \qquad (*)$$

We define an enumeration function p for the pure configurations:

$$p \equiv \lambda i.\ \langle pcs\ i,\ pcss\ i,\ S\ (f\ i)\rangle.$$

With $(*)$ we get $p\ 0 = ([c], [], s)$. With Lemma A.13 we sequence the configurations p to an infinite computation:

$$\forall i.\ \Gamma\vdash p\ i \rightarrow p\ (i+1).$$

Hence we have shown the left branch of the thesis. □

With this case distinction lemma on infinite computations we can show that terminating programs cause no infinite computations:

Lemma A.15 ► *If* $\Gamma\vdash c \downarrow s$ *then* $\neg\ \Gamma\vdash \langle[c],[],s\rangle \rightarrow \ldots(\infty)$.

Proof. By induction on $\Gamma\vdash c \downarrow s$ and lemma A.14. □

We continue with the other direction. From the absence of an infinite computation to termination. Analogous to the generalised big-step semantics for statement lists and continuations we define a generalised termination judgement:

Definition A.3 ► *Extended termination judgement for Simpl*

The extended termination judgement $\Gamma\vdash cs,css \Downarrow s$ *is defined inductively by the rules in Figure 2.2, where:*

$$\begin{array}{ll} \Gamma :: {'p} \rightharpoonup ({'s}, {'p}, {'f})\ com & cs :: ({'s}, {'p}, {'f})\ com\ list \\ s :: ({'s}, {'f})\ xstate & css :: ({'s}, {'p}, {'f})\ continuation\ list \end{array}$$

$$\frac{}{\Gamma\vdash[],[] \Downarrow s}\ (\textsc{Nil}) \qquad \frac{\Gamma\vdash c \downarrow s \qquad \forall t.\ \Gamma\vdash \langle c,s\rangle \Rightarrow t \longrightarrow \Gamma\vdash cs,css \Downarrow t}{\Gamma\vdash c\cdot cs,css \Downarrow s}\ (\textsc{Cons})$$

. . . .

$$\frac{\Gamma\vdash nrms,css \Downarrow Normal\ s}{\Gamma\vdash[],(nrms,\ abrs)\cdot css \Downarrow Normal\ s}\ (\textsc{ExitBlockNormal})$$

$$\frac{\Gamma\vdash abrs,css \Downarrow Normal\ s}{\Gamma\vdash[],(nrms,\ abrs)\cdot css \Downarrow Abrupt\ s}\ (\textsc{ExitBlockAbrupt})$$

$$\frac{\Gamma\vdash nrms,css \Downarrow Fault\ f}{\Gamma\vdash[],(nrms,\ abrs)\cdot css \Downarrow Fault\ f}\ (\textsc{ExitBlockFault})$$

$$\frac{\Gamma\vdash nrms,css \Downarrow Stuck}{\Gamma\vdash[],(nrms,\ abrs)\cdot css \Downarrow Stuck}\ (\textsc{ExitBlockStuck})$$

Figure A.2: Extended termination judgment for Simpl

We prove that $\neg\ \Gamma\vdash \langle cs,css,s\rangle \rightarrow \ldots(\infty)$ implies $\Gamma\vdash cs,css \Downarrow s$ in two steps. First, we provide a well-founded relation on configurations that can be derived from $\neg\ \Gamma\vdash \langle cs,css,s\rangle \rightarrow \ldots(\infty)$. Then we prove termination by well-founded induction on this relation.

Definition A.4 ►

$$(\prec_c^\Gamma) \equiv \{(c_2, c_1).\ \Gamma\vdash c \rightarrow^* c_1 \wedge \Gamma\vdash c_1 \rightarrow c_2\}$$

We have $c_2 \prec^{\Gamma}_{c} c_1$, if configuration c_1 is reachable from the initial configuration c and c_2 is reachable from c_1 by a single step. For a termination computation c_2 is "nearer" to the end.

If $\neg\, \Gamma\vdash \langle cs,css,s\rangle \rightarrow \ldots(\infty)$ *then* $wf\ (\prec^{\Gamma}_{\langle cs,\ css,\ s\rangle})$. ◂ Lemma A.16

Proof. We do a proof by contradiction. According to Lemma 3.16 we assume that there is an infinite descending chain, i.e. there is an enumeration f of configurations such that:

$$\forall i.\ \Gamma\vdash \langle cs,\ css,\ s\rangle \rightarrow^{*} f\ i \wedge \Gamma\vdash f\ i \rightarrow f\ (i + 1).$$

By reflexive transitive closure induction we show that we can construct an enumeration g that begins with the initial configuration $g\ 0 = \langle cs,\ css,\ s\rangle$ such that $\forall i.\ \Gamma\vdash g\ i \rightarrow g\ (Suc\ i)$. This contradicts the assumption that there is no infinite computation: $\neg\, \Gamma\vdash \langle cs,css,s\rangle \rightarrow \ldots(\infty)$. □

If $\prec^{\Gamma}_{\langle cs,\ css,\ s\rangle}$ *is well-founded:* $wf\ (\prec^{\Gamma}_{\langle cs,\ css,\ s\rangle})$*, and configuration* $\langle cs_1,\ css_1,\ s_1\rangle$ *is reachable:* $\Gamma\vdash \langle cs,\ css,\ s\rangle \rightarrow^{*} \langle cs_1,\ css_1,\ s_1\rangle$*, then it is also terminating:* $\Gamma\vdash cs_1,css_1 \Downarrow s_1$. ◂ Lemma A.17

Proof. By well-founded induction on the relation $\prec^{\Gamma}_{\langle cs,\ css,\ s\rangle}$ and the configuration $\langle cs_1,\ css_1,\ s_1\rangle$. As induction hypothesis we get that all configurations $\langle cs_2,\ css_2,\ s_2\rangle$, such that $\Gamma\vdash \langle cs_1,\ css_1,\ s_1\rangle \rightarrow \langle cs_2,\ css_2,\ s_2\rangle$, are terminating: $\Gamma\vdash cs_2,css_2 \Downarrow s_2$. By case distinction on configuration $\langle cs_1,\ css_1,\ s_1\rangle$ and executing the next step symbolically according the small-step semantics, it is straightforward to construct $\Gamma\vdash cs_1,css_1 \Downarrow s_1$ from the hypothesis. □

By combining Lemmas A.16 and A.17 and instantiating $\langle cs_1,\ css_1,\ s_1\rangle$ with the initial configuration $\langle cs,\ css,\ s\rangle$ we get:

If $\neg\, \Gamma\vdash \langle cs,css,s\rangle \rightarrow \ldots(\infty)$ *then* $\Gamma\vdash cs,css \Downarrow s$. ◂ Lemma A.18

Specialising cs to $[c]$ and css to $[]$ we arrive at:

If $\neg\, \Gamma\vdash \langle [c],[],s\rangle \rightarrow \ldots(\infty)$ *then* $\Gamma\vdash c \downarrow s$. ◂ Lemma A.19

Together with Lemma A.15 we have proven the equivalence Theorem 3.21 for termination and the absence of infinite computations:

$$\Gamma\vdash c \downarrow s = (\neg\, \Gamma\vdash \langle [c],[],s\rangle \rightarrow \ldots(\infty))$$

◂ Theorem A.20 *Termination iff no infinite computation*

List of Figures

Bibliography

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[3] Pierre America and Frank de Boer. Proving total correctness of recursive procedures. *Information and Computation*, 84:129–162, 1990.

[4] Hasan Amjad. *Combining model checking and theorem proving*. PhD thesis, Cambridge University, 2004.

[5] Krzysztof Apt. Ten Years of Hoare's Logic: A Survey — Part II: Nondeterminism. *Theoretical Computer Science*, 28:83–109, 1984.

[6] Krzysztof Apt and Gordon Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33:724–767, 1986.

[7] Krzysztof R. Apt. Ten years of Hoare's Logic: A survey — Part I. *ACM Trans. Programming Languages and Systems*, 3(4):431–483, October 1981.

[8] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 1991.

[9] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. submitted for publication, 2005. Available at `http://www.andrew.cmu.edu/user/avigad/`.

[10] Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs: International Workshop, TYPES 2003, Torino, Italy, April 30–May 4, 2003, Selected Papers*, number 3085 in LNCS, pages 34–50. Springer, 2004.

[11] Damian Barsotti, Leonor Prensa Nieto, and Alwen Tiu. Verification of clock synchronization algorithms: Experiments on a combination of deductive tools. In *Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'05)*. To appear.

[12] Bernhard Beckert and Gerd Beuster. Formal specification of security-relevant properties of user interfaces. In *Proceedings, 3rd International Workshop on Critical Systems Development with UML, Lisbon, Portugal*, Munich, Germany, 2004. TU Munich Technical Report TUM-I0415.

[13] M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing for higher-order imperative functions. In *The 10th ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*. To appear.

[14] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[15] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jerome Feret, Laurent Mauborgne, Antoine Mine, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In Ron Cytron and Rajiv Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI'03)*, pages 196–207. ACM Press, 2003.

[16] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Trans. Softw. Eng.*, 21(10):785–798, 1995.

[17] Richard Bornat. Proving pointer programs in Hoare Logic. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC 2000)*, volume 1837 of *LNCS*, pages 102–126. Springer, 2000.

[18] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Design Automation Conference, 1990. Proceedings. 27th ACM/IEEE*, pages 40–45, june 1990.

[19] Randal E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[20] Rod Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 23–50. Edinburgh University Press, 1972.

[21] Robert Cartwright and Derek Oppen. Unrestricted procedure calls in hoare's logic. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 131–140, New York, NY, USA, 1978. ACM Press.

[22] Amine Chaieb. Proof-producing program analysis. Technical report, TU München, 2005. Available at `http://www4.in.tum.de/~chaieb/articles/tr/pppa.ps`.

[23] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[24] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In Chris Hankin and Igor Siveroni, editors, *Static Analysis: 12th International Symposium, SAS 2005, London, UK, September 7-9*, volume 3672 of *LNCS*, 2005.

[25] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, February 1978.

[26] Matthias Daum, Stefan Maus, Norbert Schirmer, and Mohamed Nassim Seghir. Integration of a software model checker into Isabelle. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, Lect. Notes in Art. Int. Springer, 2005. To appear.

[27] Andreas Dehne. Beweiserzeugende Programmanalyse: Intervallanalyse. Master's thesis, Technische Universität München, 2005.

[28] W. DeRoever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, New York, NY, USA, 1999.

[29] Edsger W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[30] Peter Naur (ed.), J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauer, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6(1):1–17, January 1963.

[31] V. K. Pisini et al. Formal hardware verification by integrating HOL and MDG. In *ACM Great Lakes Symposium on VLSI*, pages 23–28, 2000.

[32] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[33] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.

[34] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Mie Barnett, editors, *Formal Methods and Software Engineering 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004*, volume 3308 of *LNCS*. Springer, 2004.

[35] Robert W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium in Applied Mathematics*, volume 19, pages 19–32, 1967.

[36] Nissim Francez. *Program Verification*. Addison-Wesley, 1992.

[37] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In Hurd and Melham [52], pages 1–16.

[38] Georges Gonthier. A computer-checked proof of the four colour theorem. Technical report, Microsoft Research Cambridge, 2005.

[39] M.C.J. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer, 1989.

[40] Michael J.C. Gordon. From LCF to HOL: a short history. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Essays in Honour of Robin Milner*. MIT Press, 1998.

[41] Gerald Arthur Gorelick. A complete axiomatic system for proving assertions about recursive and non-recursive programs. Technical Report 75, Dept. of Computer Science, Univ. of Toronto, 1975.

[42] James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

[43] David Gries and Gary Levin. Assignment and procedure call proof rules. *ACM Trans. Programming Languages and Systems*, 2(4):564–579, 1980.

[44] J.V. Guttag, J.J. Horning, and R.L. London. A proof rule for euclid procedures. In Erich J. Neuhold, editor, *Formal Description of Programming Concepts: Proceedings of the Working Conference, St. Andrews N.B. Canada, August, 1977*, pages 211–220, New York, NY, USA, 1978. Elsevier Science Inc.

[45] David Harel. *First-Order Dynamic Logic*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1979.

[46] John Harrison. Formalizing Dijkstra. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98*, volume 1497 of *LNCS*, pages 171–188, Canberra, Australia, 1998. Springer.

[47] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:567–580,583, 1969.

[48] C.A.R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Semantics of algorithmic languages*, volume 188 of *Lecture Notes in Mathematics*, pages 102–116. Springer, 1971.

[49] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[50] Peter V. Homeier. *Trustworthy Tools for Trustworthy Programs: A Mechanically Verified Verification Condition Generator for the Total Correctness of Procedures*. PhD thesis, Department of Computer Science, University of California, Los Angeles, 1995.

[51] Marieke Huisman. *Java program verification in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, 2000.

[52] J. Hurd and T. Melham, editors. *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 2005*, volume 3603 of *LNCS*. Springer, 2005.

[53] Bart Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming*, 58:61–88, 2004.

[54] Kathleen Jensen, Niklaus Wirth, Andrew B. Mickel, and James F. Miner. *PASCAL - User Manual and Report,ISO Pascal Standard*. Springer, 1991.

[55] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 2nd edition, 1990.

[56] Jeffrey J. Joyce and Carl-Johan H. Seger. Linking BDD-based symbolic evaluation to interactive theorem-proving. In *Proceedings of the 30th international conference on Design automation*, pages 469–474, New York, NY, USA, 1993. ACM Press.

[57] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., 1988.

[58] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Programming Languages and Systems*. To appear.

[59] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004.

[60] Thomas Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, Department of Computer Science, University of Edinburgh, 1998. Report ECS-LFCS-98-392.

[61] Thomas Kleymann. Hoare Logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.

[62] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *3rd International Conference on Software Engineering and Formal Methods (SEFM 2005), 5-9 September 2005, Koblenz, Germany*, 2005.

[63] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, 1995.

[64] R.L. London, J.V. Guttag, J.J. Horning, B.W. Lampson, J.G. Mitchell, and G.J.Popek. Proof rules for the programming language euclid. *Acta Informatica*, 10:1–26, 1978.

[65] Claude Marché and Christine Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In Hurd and Melham [52], pages 261–277.

[66] Laurent Mauborgne. Astrée: Verification of absence of runtime error. In René Jacquart, editor, *Building the information society (WCC'04)*, pages 385–392. Kluwer, 2004.

[67] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *Automated Deduction — CADE-19*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.

[68] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 2005. To appear.

[69] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In David Basin and Michaël Rusinowitch, editors, *Automated Reasoning — Second International Joint Conference, IJCAR 2004*, LNAI 3097, pages 372–384. Springer, 2004.

[70] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *LNCS*, pages 63–77. Springer, 2000.

[71] M.J.C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer, Berlin.

[72] Carroll Morgan. *Programming from specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[73] J.H. Morris. Comments on "procedures and parameters". Undated and unpublished.

[74] Wolfgang Naraschewski and Markus Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs'98*, volume 1479 of *LNCS*. Springer, 1998.

[75] Greg Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980.

[76] H. Nielson and F. Nielson. *Semantics with Applications*. Wiley and Sons, Chichester, 1992.

[77] Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic (CSL 2002)*, volume 2471 of *LNCS*, pages 103–119. Springer, 2002.

[78] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[79] Tobias Nipkow. Structured proofs in Isar/HOL. In *TYPES 2002*, volume 2646 of *LNCS*. Springer, 2002. `http://isabelle.in.tum.de/docs.html`.

[80] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. `http://www.in.tum.de/~nipkow/LNCS2283/`.

[81] Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[82] Russell O'Connor. Essential incompleteness of arithmetic verified by Coq. In Hurd and Melham [52], pages 261–277.

[83] David von Oheimb. Hoare logic for mutual recursion and local variables. In C. Pandu Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science (FST&TCS)*, volume 1738 of *LNCS*, pages 168–180. Springer, 1999.

[84] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[85] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P. Lindsay, editors, *Formal Methods Europe (FME 2002)*, volume 2391 of *LNCS*, pages 89–105. Springer, 2002.

[86] Ernst-Rüdiger Olderog. On the notion of expressiveness and the rule of adaptation. *Theoretical Computer Science*, 24:337–347, 1983.

[87] Veronika Ortner. Verification of BDD Algorithms. Master's thesis, Technische Universität München, 2004. `http://www.veronika.langlotz.info/`.

[88] Veronika Ortner and Norbert Schirmer. Verification of BDD normalization. In Hurd and Melham [52], pages 261–277.

[89] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.

[90] Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828. Springer, New York, NY, USA, 1994.

[91] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

[92] Lawrence C. Paulson. The relative consistency of the axiom of choice — mechanized using Isabelle/ZF. *LMS Journal of Computation and Mathematics*, 6:198–248, 2003. `http://www.lms.ac.uk/jcm/6/lms2003-001/`.

[93] Cees Pierik and Frank S. de Boer. A syntax-directed Hoare Logic for object-oriented programming concepts. In *Formal Methods for Open Object-Based Distributed Systems 2003*, volume 2884 of *LNCS*, pages 64–78. Springer, 2003.

[94] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, January 11-13, 2004, Proceedings*, volume 2937 of *LNCS*. Springer, 2004.

[95] Andreas Podelski and Andrey Rybalchenko. Transition predicate abstraction and fair termination. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 132–144, New York, NY, USA, 2005. ACM Press.

[96] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP '99)*, volume 1576 of *LNCS*, pages 162–176. Springer, 1999.

[97] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[98] Leonor Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002.

[99] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *LNCS*, pages 84–97, Liege, Belgium, jun 1995. Springer-Verlag.

[100] Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Michael Balser. Structured specifications and interactive proofs with KIV. In W. Bibel and P.H.Schmitt, editors, *Automated Deduction — A Basis for Applications. Volume II*, volume 9 of *Applied Logic Series*, pages 13–40. Kluwer, 1998.

[101] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 55–74, 2002.

[102] Norbert Schirmer. Java definite assignment in in Isabelle/HOL. In Susan Eisenbach, Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter, and Erik Poll, editors, *Formal Techniques for Java-like Programs 2003 (Proceedings)*. Chair of Software Engineering, ETH Zürich, 2003. Technical Report 108.

[103] Norbert Schirmer. Analysing the Java package/access concepts in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 16(7):689–706, 2004.

[104] Thomas Schreiber. Auxiliary variables and recursive procedures. In *TAPSOFT'97: Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 697–711. Springer, 1997.

[105] Sunae Seo, Hongseok Yang, and Kwangkeun Yi. Automatic construction of hoare proofs from abstract interpretation results. In *The First Asian Symposium on Programming Languages and Systems,LNCS Vol. 2895*, pages 230–245, Beijing, 2003. Springer.

[106] Stefan Sokołowski. Total correctness for procedures. In *Mathematical Foundations of Computer Science (MFCS)*, volume 53 of *LNCS*, pages 475–483. Springer, 1977.

[107] Kurt Stenzel. A formally verified calculus for full Java Card. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*, volume 3116 of *LNCS*, pages 491–505. Springer, 2004.

[108] Kurt Stenzel. *Verification of Java Card Programs*. PhD thesis, Universität Augsburg, 2005.

[109] Harvey Tuch and Gerwin Klein. Verifying the L4 virtual memory subsystem. In Gerwin Klein, editor, *Proc. NICTA Workshop on OS Verification 2004*, 2004. ID: 0401005T-1.

[110] Harvey Tuch and Gerwin Klein. A unified memory model for pointers. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, Lect. Notes in Art. Int. Springer, 2005. To appear.

[111] J. von Wright, J. Hekanaho, P. Luostarinen, and T. Långbacka. Mechanizing some advanced refinement concepts. *Formal Methods in System Design*, 3:49–81, 1993.

[112] Philip Wadler. The essence of functional programming. In *Proc. 19th ACM Symp. Principles of Programming Languages*, 1992.

[113] Tjark Weber. Using a SAT solver as a fast decision procedure for propositional logic in an LCF-style theorem prover. In Joe Hurd, Edward Smith, and Ashish Darbari, editors, *Theorem Proving in Higher Order Logics – 18th International Conference, TPHOLs 2005, Oxford, UK, August 2005, Emerging Trends Proceedings*, pages 180–189, Oxford, UK, August 2005. Oxford University Computing Laboratory, Programming Research Group. Research Report PRG-RR-05-02.

[114] Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics, TPHOLs'99*, volume 1690 of *LNCS*, pages 167–183. Springer, 1999.

[115] Markus Wenzel. Miscellaneous Isabelle/Isar examples for higher order logic. Isabelle/Isar proof document, 2001.

[116] Markus Wenzel. *Isabelle/Isar — A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002.

[117] Martin Wildmoser, Amine Chaieb, and Tobias Nipkow. Bytecode analysis for proof carrying code. In *Proceedings of the 1st Workshop on Bytecode Semantics, Verification and Transformation, Electronic Notes in Computer Science*, 2005.

[118] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[119] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

Index