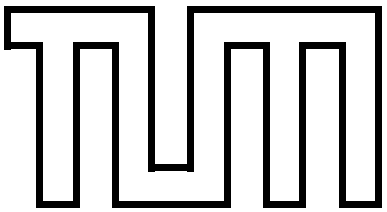
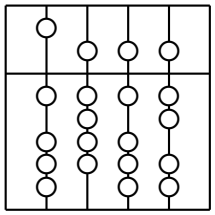




Verified Proof Carrying Code

Martin Wildmoser

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

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

Verified Proof Carrying Code

Martin Wildmoser

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. Uwe Baumgarten

Prüfer der Dissertation: 1. Univ.-Prof. Tobias Nipkow, Ph.D.

2. Univ.-Prof. Martin Hofmann, Ph.D.
(Ludwig-Maximilians-Universität München)

Die Dissertation wurde am 3.11.2005 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.5.2006 angenommen.

Kurzfassung

Proof Carrying Code (PCC) ist eine Technik zum Ausschluss von Sicherheitsfehlern in Maschinencode. Statt Laufzeittests durchzuführen, wird statisch ein Beweis (Zertifikat) geprüft. Um zu garantieren, dass ein solches System nur sicheren Code akzeptiert, formalisieren und verifizieren wir PCC in Isabelle/HOL, einem Beweissystem für höherstufige Logik. Wir beweisen, dass zertifizierter Code sicher ist, und unter welchen Voraussetzungen sich sicherer Code zertifizieren lässt. Der Hauptbeitrag ist ein generischer Verifikationsbedingungsgenerator (VCG), den wir für eine Teilsprache von Java-Bytecode instanziieren. Dieser VCG inspiziert Bytecode, der mit Formeln einer eigens geschaffenen Zusicherungssprache annotiert ist, und liefert Beweisverpflichtungen, die arithmetischen Überlauf und falsche Annotationen ausschließen. Annotationen können wir manuell oder mittels angebundener Analysatoren für Typen und Intervalle einfügen. Zur Gewinnung und Überprüfung der Zertifikate setzen wir wiederum Isabelle/HOL ein.

II

Abstract

Proof Carrying Code (PCC) is a technique to exclude safety errors in low level code. Instead of runtime tests, it statically checks a proof of safety (a certificate) attached to the code. To guarantee that PCC only accepts safe code, we formalise and verify it in Isabelle/HOL, an interactive theorem prover for higher order logic. In an abstract framework we identify key components and their interfaces, specify requirements and prove theorems stating that accepted code is safe and under what conditions safe code can be certified. Our main contribution is a generic verification condition generator (VCG), which inspects code and emits proof obligations. By adjusting parameters, we instantiate this VCG to a Java-like bytecode language with objects, methods and exceptions. Bytecode with annotations in a first order assertion logic can be certified not to cause arithmetic overflow. To annotate code we integrate bytecode analysers for intervals and types. Finally, Isabelle's facilities for code generation, proof production and proof checking enable us to turn our formalisation into a runnable prototype.

Acknowledgements

I very much thank Tobias Nipkow for supervising this thesis, offering me a position in his group and giving me the opportunity to work on a wonderful topic.

I also want to thank Prof. Martin Hofmann for acting as referee and Prof. Manfred Broy together with his staff for excellent working conditions.

I am deeply indebted to Stefan Berghofer, Amine Chaieb and Gerwin Klein, whose work and participation on the project gave me a solid ground to build on.

I am very grateful to Jorge Fox, Norbert Schirmer and Tjark Weber for many inspiring and joyful discussions on the topic and other things. I also thank them, as well as the collegues mentioned above, for reading and commenting on drafts of this thesis.

Finally, I thank the users and developers of Isabelle in Munich not only for their technical advice, but in particular for the friendly atmosphere. Many thanks to Clemens Ballarin, Gertrud Bauer, Florian Haftmann, Alexander Krauss, Steven Obua, Sebastian Skalberg, Martin Strecker, and Markus Wenzel.

Contents

1 Introduction

This chapter introduces and motivates Proof Carrying Code. We take a glimpse at related work and assess each approaches merits and drawbacks from our perspective. Finally, we give an overview on our own approach and the structure of this thesis.

1.1 Proof Carrying Code

Proof Carrying Code (PCC), first proposed by Necula and Lee [67, 68], is a scheme for executing untrusted code safely. Fig. 1.1 shows the architecture of a PCC system. The code producer is on the left, the code receiver on the right. Both sides run a verification condition generator (VCG). The VCG inspects every instruction and emits proof obligations ensuring safety. For this purpose it expects certain positions in the code to be annotated with assertions, e.g. loop or function entry points. The logic used for assertions and for the proof of safety is the *safety logic*, the property that is shown about the program is the *safety policy*.

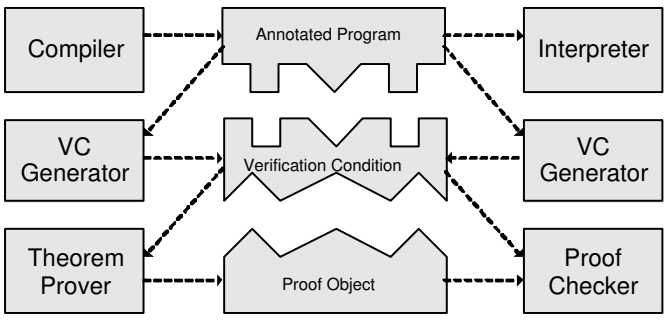


Figure 1.1: PCC Architecture

It is the responsibility of the producer to generate the annotations and a proof for the formula the VCG constructs. Annotations, program and proof are then transmitted to the code receiver. The code receiver runs the VCG again and uses a proof checker to verify that the proof indeed fits the generated verification condition. Proof checking is

usually much simpler and more efficient than proof searching. Hence, the main burden lies on the shoulders of the producer. However, once proper annotations and a correct proof are found, the code can be installed and run without further runtime checks. Since the static check needs to be done only once, this makes PCC interesting for safe code that should run efficiently. Apart from web applets and device drivers mobile code applications are still to come. Interesting possibilities open with ubiquitous computing. Small devices (smart cards, pdas, phones, ...) coming from different vendors and following different standards could teach each other how to communicate by sending each other drivers. Even without the mobile aspect, PCC is interesting for high risk software, such as control code for air planes or rockets. Since one verifies the code that actually flies, one does not have to trust a compiler. In addition, one knows that certain errors will never occur and can thus avoid error recovery or self destruction.

1.2 Contributions

This thesis makes various contributions to the field of low level software verification and applications of interactive theorem proving. A detailed discussion of strengths and weaknesses follows in §9.

(1) An abstract framework for a Verification Condition Generator (VCG), stating explicit requirements and providing proofs of correctness and completeness.

(2) Instantiations of verified VCGs for Java-like bytecode with support for objects, inheritence, dynamic method invocations and exceptions.

(3) An assertion logic for Java-like bytecode, which abstracts machine states, states safety policies and expresses verification conditions.

(4) Integration of trusted and untrusted program analysers for verification.

(5) A runnable PCC system for Java-like bytecode supporting advanced safety policies, such as arithmetic overflow.

1.3 Related Work

In this section we shortly summarise other people's work on proof carrying code. The arrangement of subsections is roughly chronological based on the first publication from the corresponding authors. We try to make the different approaches comparable by highlighting three aspects. The underlying programming language, the safety policy

and the safety logic. In our framework these will be the main factors determining a PCC system. Finally, we try to assess each approaches merits and drawbacks from our perspective. We also try to visualise each approach using data flow diagrams. Boxes with round corners denote components taking data represented in boxes with rectangular corners as input or producing such data as output. Arrows indicate how the data flows. Apart from that we sometimes use ovals connected to components via dotted lines. These denote parameters influencing the components behaviour.

1.3.1 Touchstone

The Touchstone system, shown in Fig. 1.2, is the main result of Necula's pioneering work on PCC [68]. Programs written in Safe-C, a type-safe subset of C, are compiled to assembly code. The compiler also annotates loops, function entry and exit points with types for registers, stack and heap positions. A formula annotated to some code position symbolically approximates the states under which this position can be reached at runtime. The compiler generates these formulas from source language types found in variable and function declarations. It knows how values of these types are represented in the machine and can choose corresponding machine level types. The compiler also performs various code optimisations and simultaneously adjusts the annotations. The annotated assembly program is then symbolically executed by a VCG. Starting with formulas for function entry points the VCG follows the control flow until a return instruction occurs or an annotated position is reached twice. Whenever it passes a control flow edge it transforms the formula, which abstracts the current state, using a strongest postcondition operator. Depending on the instruction to be executed next it also drops a safety condition as proof obligation. One has to prove that this safety condition follows from the current state formula. All proof obligations together make up the verification condition, a formula that ensures the safety policy.

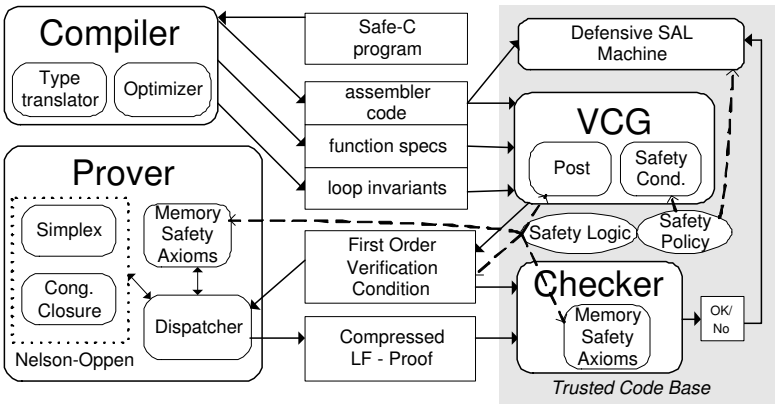


Figure 1.2: Touchstone PCC

Programming Language

The symbolic execution is done on a RISC-like assembly language, called SAL. Existing machine languages can be supported by translating their code to SAL and verifying the translation. Necula's PhD thesis [68] mentions translators for DecAlpha and x86

assembly code. With only 13 instructions SAL provides the essential operations needed to run a machine with a bank of custom and special (stack pointer, return address) registers and heap memory. There are instructions to initialise, copy, perform arithmetic or logical operations on these registers. To organise the control flow there are direct and conditional jumps as well as function calls and returns. Finally, there are instructions to write to and read from the memory. Unlike typical assembly languages SAL distinguishes memory and stack access with syntactically different instructions. This design choice is helpful to express and verify the safety policy, because different rules can be designed for stack and normal memory access.

Safety Policy

Programs are safe if they access the memory in a regulated way. Stack accesses are only allowed within the frame of the current method, and heap accesses must be safe with respect to a memory safety policy expressed by checking functions *safeRD* and *safeWR*. These checks are evaluated by a defensive SAL machine, which gets stuck if they are violated. Programs are safe if the defensive machine never gets stuck before returning from the main function.

Safety Logic

The safety policy is verified by proving the verification condition for a program. A pen-and-paper soundness proof [68] guarantees that only safe programs have provable verification conditions. The logic employed to formulate and prove verification conditions is called safety logic. In [68] this is a first order predicate logic with extensions for memory safety. The latter is supported by special predicate symbols *safeRd* and *safeWr*, whose semantic counterparts are the corresponding checks built into the SAL semantics. These predicates can be derived by using axiomatic typing rules together with ordinary natural deduction rules. The verification conditions are hereditary Harrop formulas and can thus be solved by a Prolog [23] like verification system. For reasons of efficiency Necula [68] has implemented his own theorem prover. As Fig. 1.2 shows this prover combines decision procedures for linear inequalities (Simplex [38]) and congruence closure [44, 71, 12] in a Nelson-Oppen [70] architecture. Non-convex theories, such as the memory safety typing rules, are integrated via a top-level Dispatcher. All decision procedures certify their work by emitting LF proof objects. The dispatcher combines these to a full proof of the verification condition. This proof is then sent to the consumer, who again runs the VCG on the received code and checks, for example with Twelf's [77] LF

proof checker, whether the proof fits the resulting formula.

Conclusion

Touchstone PCC is highly efficient. The transmitted proofs are relatively small thanks to the high amount of tuning in the proof producing decision procedures, efficient proof representations using oracle strings [69] and a VCG optimised for SAL and memory safety. Primitive checks, such as ensuring that control flow does not fall out of the code range, are already performed by the VCG when it scans the code. The downside of Touchstone is that it involves complex trusted components, such as a type system with axiomatic rules for memory safety and the VCG, a program of about 23k lines of code [57]. Although these components have been verified by pen and paper any flaw in their implementation can compromise safety. For example the Special-J system [33], which compiles Java to x86, showed a critical leak in its type axioms found by League *et al.* [57].

1.3.2 Foundational Proof Carrying Code

Foundational Proof Carrying Code (FPCC) [6] aims at proving safety with a minimal trusted code base. It avoids trusted components like type-specific axioms or a VCG. Instead, it defines the operational semantics of machine code in higher order logic (HOL) using only foundational mathematics such as sets and functions. The safety policy is defined as a defensive machine where programs get stuck when unsafe. This can be directly expressed and proven in HOL. The consumer can check safety using a higher order proof checker, which are typically small and simple programs. The structure of the safety proof and the safety logic is up to the producer, who can employ any type system or verification condition generator as long as soundness proofs are also transmitted.

In reality FPCC systems turn out to be more complicated than Fig. 1.3 suggests. The problem seems to be that foundational proofs are hard to construct in general. Very often FPCC targets type safety and uses a type system to structure the proof. Fig. 1.4 shows what happened to Appel's idea in the following years. Instead of giving a direct safety proof the compiler now emits type annotations. These can be seen as an indirect safety proof. A type system, with machine checked soundness meta-theorems, guarantees that well typed programs do not go wrong. The consumer now runs two different kind of checkers. One checks the meta-theorems of the type system, and one checks that the received code types well. If we regard the type system as a part of the safety proof in Fig. 1.3 this setup is still compatible with the original idea of FPCC.

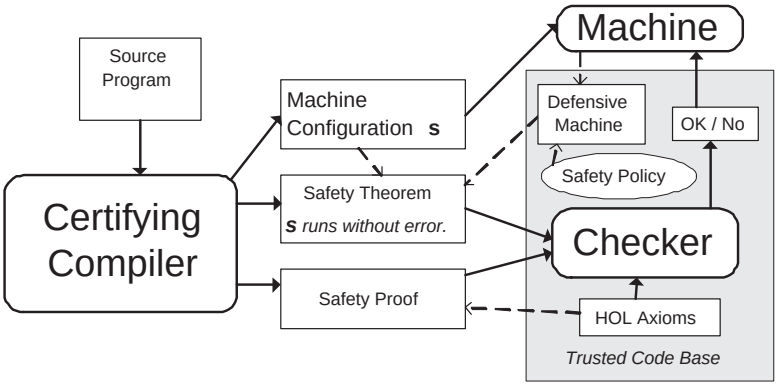


Figure 1.3: Foundational PCC

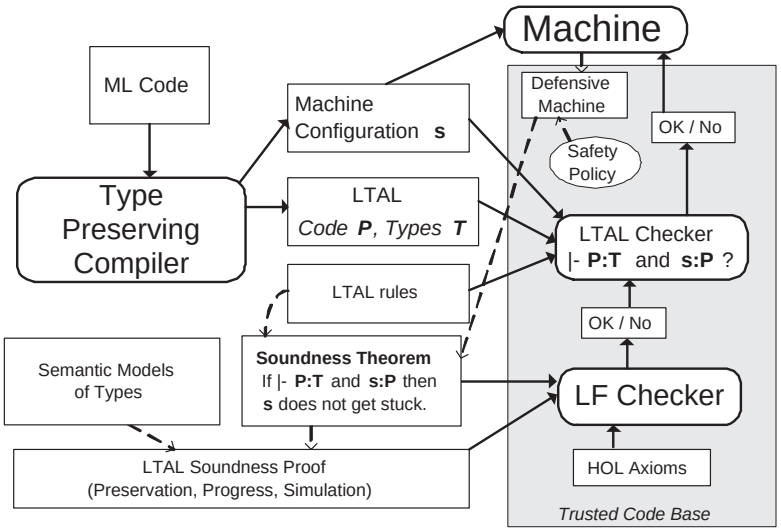


Figure 1.4: Foundational PCC in practice

Programming Language

Starting with [64] much research went into how assembly code can be type-checked and a wide variety of typed assembly languages turned up in the following years. We will discuss some of these in the next sections. Appel *et al.* now have a PCC system that supports LTAL [31, 89], a typed assembly language whose type system is expressive

enough to translate high-level ML programs into it.

Safety Policy

In the early work on FPCC [6] only simple safety policies such as correct decoding of instruction codes are addressed. Now, with the type oriented setup [31] [89], type safety is the major goal. This means machine programs are safe if they respect the data abstractions from the source program. For example, a defensive machine for type safety would get stuck if an instruction would be applied to arguments of incompatible type, e.g. adding a boolean to a string.

Safety Logic

To prove type safety Appel and Felty [7] propose to use semantic models of types. Instead of introducing types as syntactic objects with a (possibly faulty) inference system, Appel and Felty model them semantically in Twelf [77], for example as sets of values or states. Typing rules are then formulated and proven as lemmas on these type models in HOL. Modelling types as sets of values or states [7] turned out to get unwieldy when higher order polymorphism or general recursive types come into play. The approach proposed in [7] sounds complex as it involves a formalisation of the Banach fixed point theorem on complete metric spaces. Later Appel and McAllester [8] proposed a much simpler model for recursive types based on indexed values. This model is also used in the Open Verifier [30] we discuss in §1.3.6.

Conclusion

FPCC is more trustworthy and flexible than conventional PCC. The gain in trustworthiness comes from a minimal trusted code base, that only involves the machine semantics (with built-in safety policy) and the proof checker. It is flexible, because it does not demand to use a specific type system or VCG to prove safety of programs. A common strategy is to define a type system for a particular safety policy inside the foundational logic and then to reduce safety proofs to type checking via a type soundness argument. A drawback is that the soundness proof is also part of the transmitted proof and must be checked again and again for every program. This and the fact that proof obligations are typically in HOL, which is difficult to automate, makes generating foundational proofs hard. In addition, finding suitable semantic models for advanced types (polymorphic,

recursive) is a non-trivial task. Hence, it is not surprising that existing FPCC systems only handle basic safety policies, such as instruction decoding or operand safety.

1.3.3 Syntactic Proof Carrying Code

Foundational Proof Carrying Code as proposed by Appel and Felty [7] uses semantic models of types to prove the safety policy. Before indexed values came up [8] such models turned out to be complex when higher order polymorphism or contravariant types shall be supported. Syntactic Proof Carrying code [50] is like Foundational Proof Carrying Code [6], but uses explicit syntax for types and does not require a semantics of types at all. Instead, as Fig.1.5 illustrates, the proof is gained from translating a syntax driven type derivation for the source program (typed assembly language) into a safety argument for the machine language.

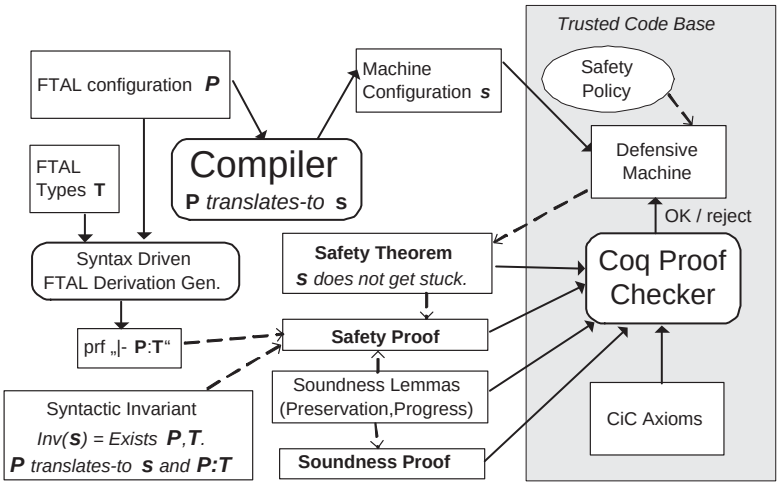


Figure 1.5: Syntactic PCC

Programming Language

The types are designed for a down sized variant of typed assembly language (TAL) [64], called Featherweight Typed Assembly Language (FTAL). This language serves as an abstract description of concrete machine programs, which are represented as a series of opcodes stored in a segment of a machine configuration. The compiler used in [50] translates an FTAL program P to an initial machine configuration s, that includes both data and code.

Safety Policy

This configuration is safe if the defensive machine does not get stuck on it, which happens when the program counter points to an invalid opcode.

Safety Logic

To prove safety Hamid *et al.* [50] follow the idea of FPCC and directly demand a proof for an instance of the safety theorem to the given configuration s (state + program code). This proof is generated by the compiler using a global inductive invariant Inv. This invariant is global because one shows that all reachable machine configurations satisfy it. It can be proven by induction on the execution semantics given by a function *step*, which computes the successor configurations. The safety policy, given in form of a predicate, must follow directly from the Invariant. This means one has to prove these three conditions:

Initial Condition $Inv\ s_0$
Preservation $\forall s.\ Inv\ s \longrightarrow Inv\ (step\ s)$
Progress $\forall s.\ Inv\ s \longrightarrow SP\ s$

The safety policy SP in [50] only holds for machine configurations where the program counter points to a memory cell containing a valid opcode. The key idea is to make the compiler and the source level type system parts of the invariant Inv, which then says that the code represented in memory is syntactically wellformed.

$Inv\ s \equiv \exists\ P : \text{program}.(P \text{ translates-to } s) \wedge \vdash_{\text{FTAL}} P$

It holds for configurations s that are translations of a well typed abstract program P. In [50] FTAL comes with a type judgement ($\vdash_{\text{FTAL}}$) and an abstract execution relation $\overset{\text{FTAL}}{\longmapsto}$. The following theorems, which are proven as meta-theorems in Coq [43], guarantee that a machine program gained by translation of a well typed FTAL program is safe. The notation $P \overset{\text{FTAL}}{\longmapsto} P'$ stands for an abstract program semantics, which transforms types instead of values.

FTAL-Preservation If $\vdash_{\text{FTAL}} P$ and $P \overset{\text{FTAL}}{\longmapsto} P'$, then $\vdash_{\text{FTAL}} P'$.
FTAL-Progress If $\vdash_{\text{FTAL}} P$, then there exists P' such that $P \overset{\text{FTAL}}{\longmapsto} P'$.
FTAL-Translation If $\vdash_{\text{FTAL}} P$ and $P \overset{\text{FTAL}}{\longmapsto} P'$ and P' translates-to s, then P' translates-to $(step\ s)$.

Once these theorems are proven, one can show the Preservation and Initial Condition by finding a well typed FTAL program that translates to the initial machine state s_0. In [50] the safety policy SP follows directly from the translation relation, which does not permit machine states with illegal instructions on its right hand side. Hence, the progress condition holds automatically because of the construction of Inv.

Conclusion

The clear advantage of the syntactic approach is, that it avoids introducing a complex semantics for the FTAL types, which include tuple types with polymorphism and recursion. Types only reside in the abstract (syntactic) domain and are dropped by the translation relation. In case of FTAL this translation converts tuple values and machine instruction sequences to machine words following a predefined memory layout. For the meta-proofs it is important that each FTAL instruction corresponds to one machine instruction. Hence, macro instructions, such as `malloc`, need to be split into various FTAL instructions. Proving safety of individual programs is made relatively simple as the FTAL typing system is purely syntax directed and easily checkable. This clearly remedies the problem of proof generation which is often seen as the major problem of FPCC. A drawback of syntactic proof carrying code is that the type system is typically designed for a particular safety policy. If one wants to change this policy one has to come up with a different type system and prove the meta-theorems for it. This makes Syntactic PCC less flexible than semantical FPCC (Fig. 1.3). In addition, the safety policy Syntactic PCC currently handles is rather basic, e.g. correct decoding of instructions. Another downside is that the gap between FTAL and machine code is quite low, only the format of instructions and the layout of structured values seem to be different. A compiler for a real high-level language to FTAL seems to be missing in the work of Hamid, Zhong *et al.*

1.3.4 Typed Assembly Languages

Morisett et al. [64, 78] introduced Typed Assembly Language as a target language for a type preserving compiler. Their motivating idea was to be able to type-check the results of a compiler rather than verifying the compiler itself. By doing this, one can ensure that data abstractions used in the source languages, e.g. lists, trees, records and so on, are respected by the assembly code. Their initial paper [64] shows how functional programs in λ^F, a call-by-value variant of System F (polymorphic λ-calculus) with products and recursion on terms, can be translated to typed assembly language. Their paper illustrates this compilation process using the factorial function as an example. Programs

pass three intermediate lambda-calculi (continuation passing style, closure conversion, stepwise allocation). Later this work has been extended to a more realistic TAL [64] targeting Intel's 80x86 architecture. Since then, a multitude of TALs, each with different modellings of instructions and typing rules, have been proposed. In the appendix of [31] one finds an overview on these. Apart from FTAL and LTAL, which we discussed in the previous chapters, there is a major body of work done by Crary *et al* [36], which this section will focus on. Fig. 1.6 outlines this approach. Crary's idea is to embed everything into a meta-logical framework and thus be able to machine-check all safety critical components.

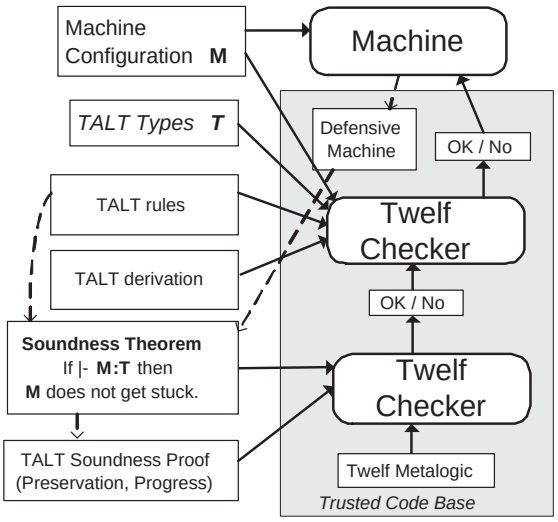


Figure 1.6: Typed Assembly Language (TALT)

Programming Language

Crary [35] formalises TALT (TAL Two) in the Twelf meta logic. TALT has similar instructions like TAL, but supports arbitrary level of indirection for passing arguments. The semantics of TALT is given by a defensive operational machine. Such a machine gets stuck, when an unsafe state is reached.

Safety Policy

The safety policy is memory safety and is built into the semantics, which is completely independent of the type system.

Safety Logic

Like in other TALs the type system plays the role of the safety logic. This type system consists of 13 different judgments, each giving constraints on different parts of the machine state: heap, program code, register files, values, and much more. For example judgement $\Psi, r1 : int, r2 : box(int), r3 : code(r1 : int, r2 : int) \vdash C$ states that code C is safe to execute if register $r1$ contains an integer, register $r2$ a pointer to an integer and register $r3$ a return address to code that is safe to execute when registers $r1$ and $r2$ contain integers. Note that TALT, as other TALs, types states and instructions simultaneously. Most of the flexibility of this type system is provided by its subtyping rules. It provides the facility for introducing or eliminating existential, universal, union, intersection or recursive types. All rules are formalised in the Twelf meta logic and a lot of specialised rules are derived therein. Soundness is guaranteed by machine-checkable proofs for progress, type preservation and a simulation theorem between the typed (static) and untyped (dynamic) semantics. These theorems look very similar to FTAL, but a major difference is that TALT is not syntax directed. Type checking is not even decidable, hence a code producer has to provide a typing derivation as proof. This derivation can then be efficiently checked by Twelf's proof checker.

Conclusion

TALT provides an elegant and expressive type system for assembly code. The entire safety argument is formalised and supported with machine-checkable proofs. The safety considerations even include a theorem stating that garbage collection preserves typeability of states. The downside is, that TALT requires typing derivations and it remains unclear where these should come from. A compiler for an appropriate source language is missing. The Twelf type checker is also larger than the simple LF type checker used by Appel [31].

1.3.5 Mobile Resource Guarantees

Mobile Resource Guarantees [81] aims at certifying bounds on memory consumption of programs. Resource-aware typing rules [52] for a functional source language ensure that programs only consume linearly many heap cells. The semantics of this functional language is defined operationally with an explicit free-list, which indicates free memory cells in the heap.

Whenever a program allocates new data, for example when a new datatype constructor is used, the semantics removes entries from the free-list. If a program releases memory, as in a destructive pattern match, the free-list grows. The amount of required and released memory is bounded by a linear expression on the input size, which is part of the typing information and derivable by a type inference algorithm. For example $x : L(L(B,1),2), 3 \vdash e : L(B,4), 5$ is a typing judgement of the system. It says that if we evaluate e in a context where x is bound to a list $[l_1, l_2, ..., l_m]$, where each l_i is a list of booleans, then a free-list of length $3 + 2m + 1 \sum_{i=1}^{m} |l_i|$ ensures that execution does not get stuck. After a successful execution with result l, the free-list is guaranteed to have length $5 + 4|l|$. If e is the body of a function with argument x a programmer specifies input and output types without resource numbers, e.g. $f : L(LB) \longrightarrow LB$. The type inference algorithm then finds out proper resource numbers. It does so by solving linear inequalities that occur in the syntax directed typing rules. The type system only works for first order functional programs with restrictions on variable sharing, but is efficient on these. MRG supports proof carrying code in its low level architecture. Typeable source programs can be compiled to a bytecode like low level language such that the type derivation can be turned into a low level safety proof.

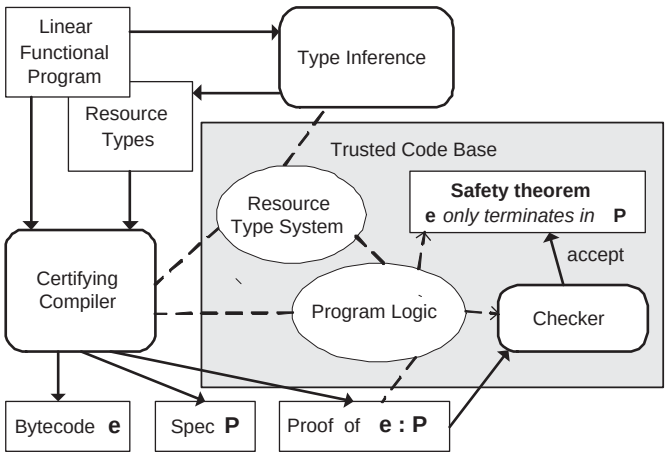


Figure 1.7: Mobile Resource Guarantees

Programming Language

The target language for MRG is Grail [19], which is a compromise between a functional language and bytecode like low level code. Assignments are simulated with Let expressions and jumps are made obsolete by turning each code block into continuation passing style. That is, jumps only occur at the end of code blocks in form of function calls. The semantics of Grail is defined with big step rules (natural semantics [72]), which closely match the semantics rules of the source language. A cost model makes the relationship to execution on a virtual machine, which is typically defined via small step rules, explicit.

Safety Policy

The safety policy for Grail is encoded into a program specification, which is a logical formula relating initial with final states. Parts of this specification define how many resources the program consumed until the final state has been reached. The specification can be automatically generated from the resource types of the source program.

Safety Logic

The safety logic comes in form of a program logic [10, 18] for Grail, whose rules closely reflect the typing rules of the source language. This makes it possible to translate a typing derivation into a safety proof. The logic is embedded into Isabelle/HOL in form of a shallow embedding. Assertions are written in VDM style [53], that is variables can refer to the current or initial state. Unlike Hoare Logic [51], where pre and postconditions are usually separated, the logic specifies the behaviour of programs in a single formula.

Conclusion

Certifying memory bounds is difficult to automate, as it involves reasoning not only on types but also on values and arithmetic expressions. In that respect MRG goes far beyond type safety. The restriction to programs with linear memory consumption is not problematic in practice. In particular not for programs running on devices with very limited memory (smart cards, pdas, phones). The sharing constraints on variables seem to be more prohibitive. There are relaxations of these restrictions[11], but these introduce further complications for the type inference. Another question is what kind

of safety guarantee MRG gives for non-terminating programs. The bytecode logic only gives partial correctness, thus non-terminating programs e satisfy every specification P.

1.3.6 Open Verifier

Instead of having a highly engineered and thus complex VCG, the Open Verifier approach [84] only maintains a small trusted core as shown in Fig. 1.8. This core consists of two subsystems, the decoder and the director.

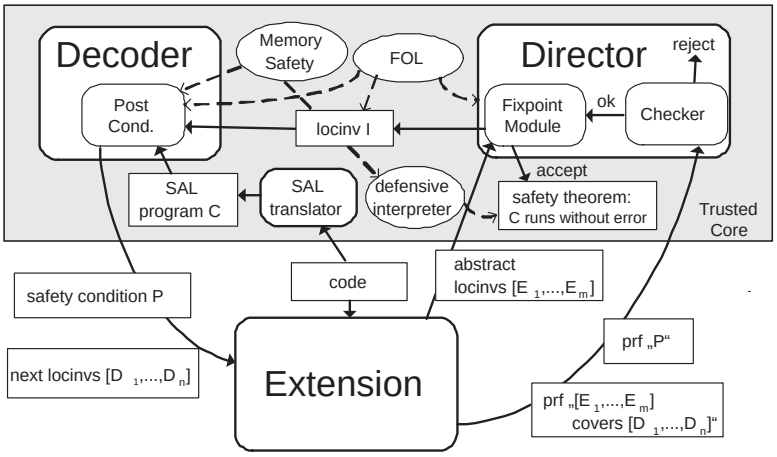


Figure 1.8: Open Verifier

The decoder maintains the abstract semantics of machine code in form of a strongest postcondition generator. It takes first order formulas in a specific format and computes the strongest postcondition and a safety condition for the current instruction. Since each formula contains a reference to the program counter it is only valid at a specific location in the program an is thus called local invariant or *locinv*. The general idea is to verify a safety policy SP by finding a set of locinvs I_k, such that the following conditions hold:

1) $I_0 \longrightarrow \bigvee_{k=1,...,n} I_k$ 2) for each $k = 1, ..., n$

a) $I_k \longrightarrow SP$

b) Post $I_k \longrightarrow \bigvee_{j=1,...,n} I_j$

A fix point module built into the core tries to find a disjunction of locinvs that covers all reachable states (2b) and implies the safety policy (2a). It starts with a disjunction that only contains I_0 and continues adding new postconditions until the collection of

locinvs becomes stable. Normally this process does not terminate, as the postcondition operator may never stop introducing new formulas. Here the extension comes into play. In order to check (2b) the extension weakens the exact postcondition $Post\ I_k$, for example, by abstracting values to their types. Instead of adding the exact postcondition to the collection, the fix point module adds the weakened formula and checks that it is indeed a weakening. To achieve this the extension has to produce a proof of $Post\ I_k \rightarrow Ext\ (Post\ I_k))$, which is then checked by the trusted core. The whole process stops when for all $k = 1, ..., n$ there is a $j = 1, ..., n$, such that $Ext(Post(I_k)) = I_j$.

Programming Language

Like Touchstone PCC [68] the Open Verifier works with SAL, the RISC-like assembly language discussed in §1.3.1. There are also instantiations of Open Verifier that support Cool, a mini object oriented language, or a variant of Typed assembly language (TAL). In both cases the languages are internally translated to SAL, the language the postcondition module understands.

Safety Policy

Similar to Touchstone PCC the Safety Policy in the Open Verifier architecture is built into the operational semantics. In case of a violation of the safety policy, which is also memory safety in [30], the program counter is set to *error*. Once programs reach this position they cannot escape from there. Hence, the safety policy SP is a predicate that checks $r_{pc} \neq error$. In addition, it also checks that code does not get overwritten, so that the post condition generator can determine the current instruction by extracting the program counter from the input locinv.

Safety Logic

The logic in which locinvs are expressed in is quite similar to the first order logic used in Touchstone. The format of formulas is fixed to a triple (Γ, R, Ψ), where Γ maps free variables to their types, R contains expressions for registers of the form $r_j = e_j$ (e_j is an arithmetic expression with variables bound in Γ), and Ψ is collection of assumptions $h : \phi$ with names h. Encoded in the logic this format stands for $\exists\ x_1 : \tau_1, ..., x_m : \tau_m \in \Gamma.\ (\bigwedge_{r=e\ \in\ R} r = e \wedge \bigwedge_{h:\phi\ \in\ \Psi} \phi)$. Note that neither the ex-

pressions e nor the formulas ϕ contain register symbols r. This greatly simplifies the computation of postconditions that would involve introducing new existentials otherwise.

Conclusion

The Open Verifier architecture is an adjustment of Touchstone PCC towards FPCC. The trusted core is a lot smaller than in Touchstone PCC, as it involves only the checker, the fix point module and the postcondition operator as well as the safety logic and policy. Parts of the Touchstone VCG have moved to the extension, but the core functionality, namely computation of weakest preconditions and adding safety conditions remains trusted. It is unclear if these parts are covered by machine-checked proofs. It is the extension which gives the Open Verifier its flexibility. Like in FPCC each extension may use its own type system or logic to prove the weakening of locinvs. In extreme cases the producer may send another extension for each piece of code. That is instead of proofs, provers are transmitted. On the other hand one may also use default provers, maybe verified in meta-theorems as extensions. This opens a great amount of flexibility, just as FPCC does. The difference between FPCC and the Open Verifier is that the latter contains a fixed trusted core providing program analysis functionality. In pure FPCC these components are missing and if they would be employed their transformations would need to be justified in the transmitted proofs. In this respect the Open Verifier is more efficient, but also a bit less flexible.

1.4 Our work

The VeryPCC project [4] aims at formalising and verifying PCC in a theorem prover. As we pointed out in the previous chapters, the main factors for PCC are the target programming language and its semantics, the safety policy and the safety logic. All these factors determine the work of the VCG, which transforms program code to proof obligations ensuring safety.
In the early PCC systems [68] (§1.3.1) these factors are hardwired into the VCG, which leads to a highly efficient, but large, complex, and rather inflexible system. Since the VCG must be trusted, large and complex handwritten code is certainly not desirable.
Eliminating the VCG, as purely foundational PCC [6] (§1.3.2) proposes, typically leads to long and complex safety proofs. Foundational proofs are rather long, because work done by the VCG, namely analysing the control flow and symbolically approximating the reachable states, becomes part of the proof. They are complex, because proving safety directly on the semantics usually involves higher order reasoning (transitive closure of

transition relation) and induction.
An alternative is to gain a trustworthy VCG by formalising it and verifying it inside a theorem prover. About the same time the Open Verifier [30] (§1.3.6) came up, we realized that this is feasible, because one can isolate a VCG's trusted core. By factoring out all parts that depend on the underlying machine semantics, safety policy and safety logic as independent parameters, one can keep the VCG compact and clear. This way the VCG not only becomes less complex, but also generic, in the sense that it can be instantiated to various PCC setups.
We propose an Isabelle/HOL [74] framework for PCC [98], whose main contribution is a generic, executable, and verified VCG. This VCG views programs as annotated control flow graphs and constructs verification conditions just as Floyd [47] proposed in the early days. That is, every annotation has to imply the weakest precondition [47, 42] for each successor annotation. This ensures that all annotations hold at runtime, provided the initial one did. Although annotations are important for verifying cyclic code, they are just a supportive concept.
Our primary concern is the validity of the safety policy. Unlike annotations, the safety policy is fixed for all programs and chosen by the consumer. Our framework expects the safety policy as a function (*safeF*), which yields a formula for a given program and position therein. This so-called safety formula expresses the conditions we want to hold, whenever this position is reached at runtime. By conjoining safety formulas with annotations we construct verification conditions in Floyd's fashion. These guarantee validity of both: annotations and safety policy. Apart from that, representing a safety policy in this way also works nicely for non-deterministic programs.
Defensive machines, which are an alternative to express safety policies, typically have problems with non-determinism. Programs that can reach states, where only some transitions are blocked, are not stuck and would thus be regarded safe. For the construction of verification conditions, two functions, also taken as parameters, play a major role. One that approximates the control flow (*succsF*) and one that computes weakest preconditions (*wpF*). Given a position, we expect *succsF* to list all possible successor positions together with so-called branch conditions. These are formulas that describe the situations when a particular successor is accessible. The second function, *wpF*, takes a control flow edge and a postcondition as input. It yields a formula that ensures that when we move along this edge at runtime, we end up in a state satisfying the postcondition.
Our framework makes all these expectations about the parameter functions explicit in form of HOL formulas expressing requirements. When one instantiates our framework by giving definitions for the parameter functions, one also has to prove that these requirements are satisfied. Relying on the requirements we prove our generic VCG correct and relatively complete.
Correctness ensures that verification conditions provable in the safety logic guarantee the safety policy. Relative completeness is less important, but desirable. It says that

when the annotations and safety formulas make up valid Hoare triples [51], then the verification conditions are tautologies. This means they are provable, when the safety logic can prove valid formulas.
Our VCG is formalised in an executable style. If the parameters are also executable, one can use Isabelle/HOL's code generator [17] to obtain a runnable ML prototype.
We have demonstrated this for a simple assembly language using Isabelle/HOL as safety logic [98, 96] and will do so for Jinja (§8). The expressiveness of HOL enables us to treat safety policies beyond the much researched type safety. We have chosen a safety policy that prohibits arithmetic overflow, a risk many programming languages, such as Java [49, 88], silently ignore. However, the instantiations in our early publications [98, 96] mainly serve as proofs of concept. They show that our framework has satisfiable requirements and can be instantiated with reasonable effort.

With Jinja PCC [97, 95] we have an instantiation that is much more realistic. For

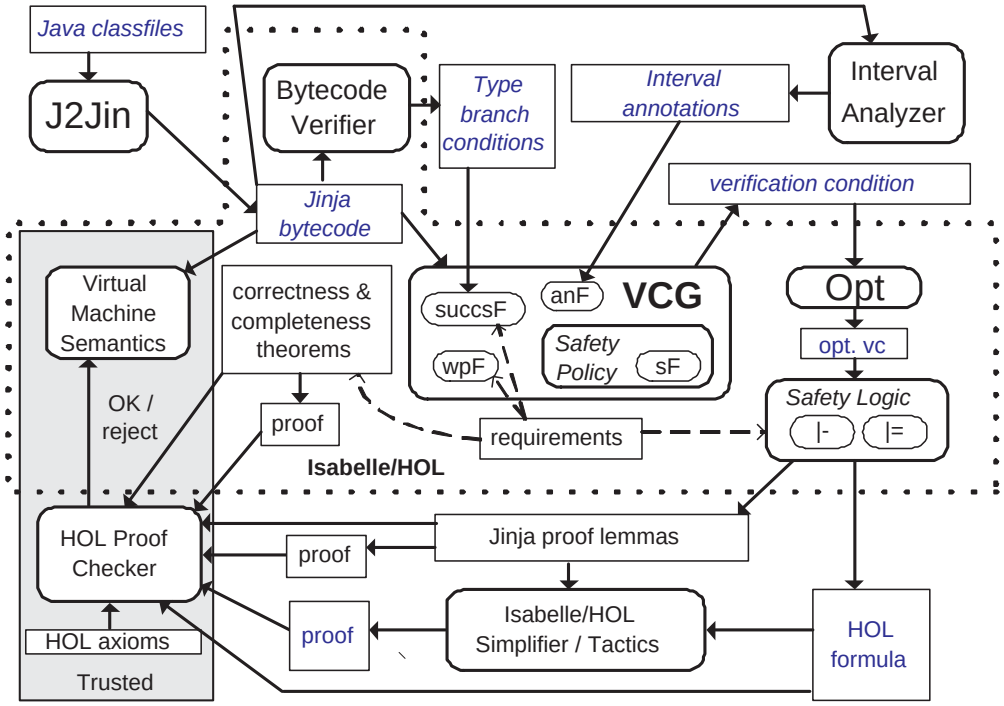


Figure 1.9: Jinja PCC

Jinja bytecode we already had a formalised and verified bytecode verifier (BCV) and virtual machine available [55]. As Fig. 1.9 illustrates both take Jinja bytecode, which we obtain from Java classfiles using a converter called `j2jin` (§3.3), as input. The BCV computes types for registers and the operand stack. Using these types together with

interval annotations we obtain from an untrusted analyser [95], our instantiated VCG constructs verification condition against arithmetic overflow.
The verification condition is an expression in an assertion language for bytecode [97].
A downside of constructing generic verification conditions is that these tend to be quite verbose. Knowing how formulas in the safety logic look like, what invariants are guaranteed by the system, and being able to identify sub formulas that are irrelevant for a particular safety policy, opens much potential for optimisations. To some extent one can optimise the instantiated VCG by defining sophisticated parameter functions. However, this usually makes proving the requirements trickier. An alternative is to analyse the resulting verification condition and postprocess it with an optimiser. Of course one has to justify this with a proof that the optimiser preserves provability of a formula. In the case of Jinja, we have done so for simple optimisations, like folding constant expressions or cutting out obviously redundant subformulas.
We have formalised the provability judgement $\vdash$ for our safety logic such that it turns assertions A into propositions $\vdash A$ in HOL. Since the verification condition vc is also an assertion, we can use Isabelle/HOL to prove it. We give the HOL proposition $\vdash vc$ to Isabelle's simplifier and decision procedures and ask Isabelle to record proof objects. Using Isabelle's proof checker we can simulate the consumer's job of validating proofs. All the components inside the dot-framed area of Fig. 1.9 are formalised in Isabelle/HOL. Most important is the VCG, for which we have correctness and completeness proofs. This makes the Jinja Virtual Machine and the Isabelle proof checker the only remaining trusted components. For the Jinja VM one has to believe that it behaves on Java programs converted by `j2jin` just as a real Java VM would behave on this program. Note that such a border between formal and real world exists in all verification projects. Also, we cannot avoid trusting Isabelle/HOL's proof checker. We need it to check safety proofs of individual programs as well as the proofs for our PCC system.

Programming Language

Jinja bytecode is a down-sized version of Java bytecode. Although it only has a few instructions, it supports the core object oriented features like object creation, inheritance, dynamic method call and exceptions. Missing are static methods and fields, non-default constructors and arrays. From Java's primitive datatypes Jinja only supports booleans, integers and references.

Safety Policy

Since Isabelle/HOL has powerful decision procedures for linear arithmetic [29] and type safety is already handled by the bytecode verifier [54], we have chosen to instantiate a safety policy for arithmetic overflow. In general arithmetic properties cannot be decided, but for many programs interval annotations suffice to prove this policy automatically.

Safety Logic

A first order expression language [97], which we designed for modelling Jinja Virtual Machine states, serves as assertion logic. We use it to express safety policies, annotations and verification conditions. Judgments for provability and validity relate assertions to HOL predicates on machine states. This enables us to use Isabelle/HOL as proof engine for safety proofs. Other tools could also be used for this purpose, as long as they emit HOL proofs our proof checker understands.

1.5 Outline

In §2 we describe our framework for PCC, which we instantiate in the following chapters. Chapter §3 formalises the syntax and semantics of Jinja bytecode and briefly discusses how we translate classfiles into that format. Then §4 introduces a formal language and semantics for annotating Jinja bytecode and expressing safety policies and verification conditions. Having all these factors set, chapter §5 defines the main parameters for our VCG: the control flow function and abstract semantics. In chapter §6 we define a safety policy against arithmetic overflow, a wellformedness checker and all the remaining parameters. This chapter also instantiates our VCG and proves that the instantiated parameters satisfy the requirements. Chapter §7 shows how different program analysers can be integrated and demonstrates this with our bytecode verifier and interval analyser. Finally §8 shows the system from a user's perspective. We demonstrate how our system simulates the different tasks PCC has for code producer and consumer. Throughout this thesis we use Isabelle/HOL as our formal meta-language. Reader's not familiar with this notation may start with the short introduction in §A.1. For more details we recommend the tutorial [74].

2 Abstract Framework

We introduce a generic framework for proof carrying code, developed and mechanically verified in Isabelle/HOL. The framework defines and verifies a verification condition generator with minimal assumptions on the underlying programming language, safety policy, and safety logic.

PCC systems mainly depend on three factors: programming language, safety policy, and safety logic. The programming language defines syntax and semantics of programs, the safety policy specifies the conditions programs must satisfy at runtime, and the safety logic provides a formal notation and a means for proving these conditions.

In classical PCC systems [68] the VCG is a major component and is affected most by these factors. Although VCGs for various PCC systems differ in detail, they all incorporate the same principle. Floyd's [47] ground breaking idea to represent concrete states as logical formulas and state transitions via formula manipulations is the essence. Hoare [51] and Dijkstra [42] later on refined this idea to structured languages.

In our formalisation of PCC we capture this essence in form of an abstract framework for a VCG. As shown in Fig. 2.1 it is the job of the VCG to reduce programs to logical formulas that are provable only if the program is safe.

In our framework we define a generic VCG, which one can instantiate to different programming languages, safety policies and logics by adjusting its parameters. An instantiation has to provide definitions for the parameter types and functions. Our framework only declares these elements and makes requirements on them.

Based on these requirements the framework provides abstract proofs of soundness and relative completeness for the VCG, which automatically carries over to all of its instantiations. The instantiation only has to provide proofs that the parameters it defines satisfy the framework's requirements.

As shown in Fig. 2.2, our framework consists of various Isabelle/HOL theories. Each theory is a box and arrows point at its parents.

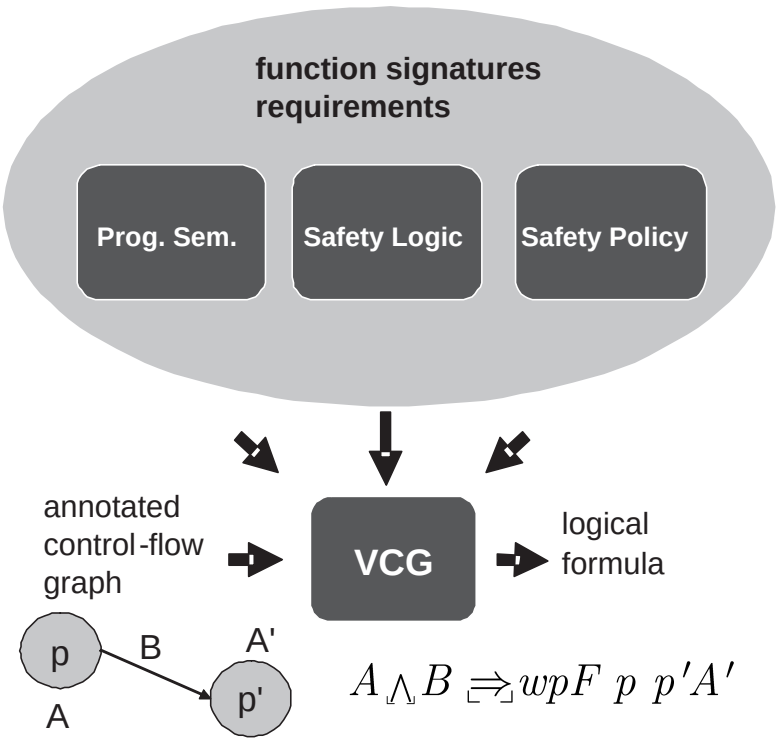


Figure 2.1: VCG - what it does and what it depends on.

2.1 Program Semantics

The first important factor of any PCC system is the programming language. Since safety properties typically only concern the execution of programs, our framework does not care about program syntax. We model this by using a type variable *'prog* to stand for program representations Π. To model execution we use state transition systems, which we formalise as an Isabelle/HOL locale called *Semantics*.

locale *Semantics* =
fixes *initS* :: *'prog* ⇒ (*'pos* × *'mem*) *set*
fixes *effS* :: *'prog* ⇒ ((*'pos* × *'mem*) × (*'pos* × *'mem*)) *set*

Given some program Π we identify the set of initial states with *initS* Π. The transition relation *effS* Π pairs each state with its immediate successors. This is what the literature calls a *small step semantics*. It is commonly used in safety analysis, where not only initial and final states matter, but in particular intermediate ones. We model states as tuples

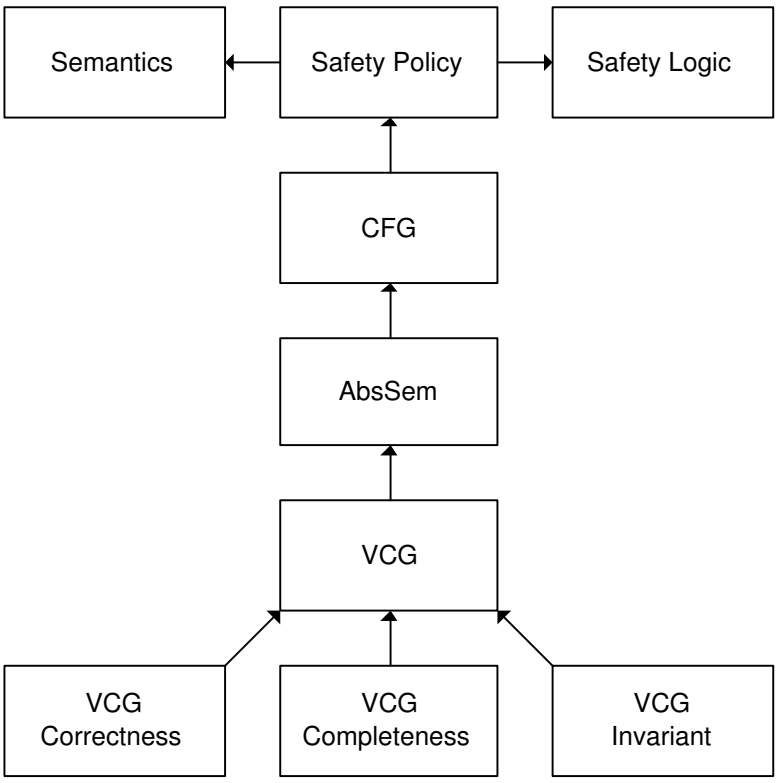


Figure 2.2: Isabelle/HOL theories of our framework

(p,m) of type $'pos \times 'mem$, where p denotes the current position in the control flow graph and m the machine's memory, e.g. heap, stack and registers. Again, these are type variables, allowing one to instantiate control flow positions and memory as one likes. For our notion of program safety (§2.3) only reachable states matter. These are states that occur in some execution sequence starting from an initial state. Using Isabelle/HOL's feature of inductive definitions we can formalise the set of reachable states as follows. First, we define the set *ReachFromIns R F I* to cover F and all elements reachable from there by taking an arbitrary number of R transitions staying inside I.

$$ReachFromIns :: ('a \times 'a)\ set \Rightarrow 'a\ set \Rightarrow 'a\ set \Rightarrow 'a\ set$$

$$\frac{a \in F}{a \in ReachFromIns\ R\ F\ I}\,\mathrm{F} \qquad \frac{a \in ReachFromIns\ R\ F\ I \qquad (a,\ b) \in R \qquad a \in I \qquad b \in I}{b \in ReachFromIns\ R\ F\ I}\,\mathrm{R}$$

In Fig. 2.3 we illustrate this reachability set. The small and shaded triangle bounds the set of initial states F. All states within the outer triangle lie inside I. The numbered

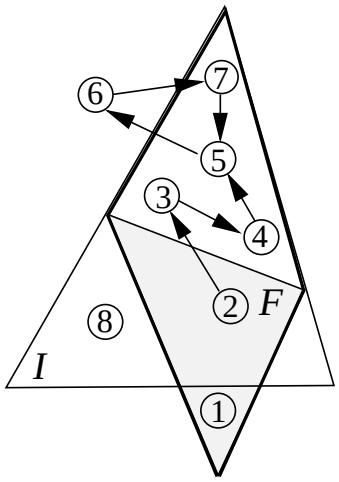


Figure 2.3: Reachability

circles represent states and the arrows R. In the depicted situation *ReachFromIns R F I* would be the set of states $S = \{1,2,3,4,5\}$. The state *1* lies in S because it belongs to F, although not to I. States such as *8* satisfy I, but are not reachable from F. State *7* is reachable from F, but only if we temporarily move to state *6* which lies outside I. This is not permitted and thus, neither *6* nor *7* belong to S.

Some further definitions don't care in who is in or out of I. In *ReachableFrom R F* we disable the set of insiders I by setting it to $\{s.\ True\}$, the set of all states. For our example in Fig. 2.3 this means *ReachableFrom R F* contains all states, except *8*.

ReachableFrom :: $('a \times 'a)$ *set* $\Rightarrow$ $'a$ *set* $\Rightarrow$ $'a$ *set*
ReachableFrom R F = *ReachFromIns R F* $\{s.\ True\}$

Using this construction, we now define *Reachables* Π, the set of states a program Π reaches.

Reachables :: $'prog \Rightarrow ('pos \times 'mem)$ *set*
Reachables Π = *ReachableFrom* (*effS* Π) (*initS* Π)

From this definition we can derive a conversion lemma using transitive reflexive closure.

Lemma 2.1 *(Semantics)* The set *Reachables* Π contains all states reachable from an initial state with an arbitrary number of state transitions.

Reachables $\Pi = \{s.\ \exists\, s_0.\ s_0 \in (initS\ \Pi) \wedge (s_0,s) \in (effS\ \Pi)^*\ \}$

Note that we have written (*Semantics*) behind Lemma 2.1. This indicates that the

lemma is proven inside the locale *Semantics*. It depends on the parameters *initS* and *effS* and all of the locale's requirements (none in this case).

2.2 Safety Logic

To specify and prove properties about programs we use a safety logic.

locale *SafetyLogic* =
fixes $\lfloor T \rfloor$, $\lfloor F \rfloor$:: *'form*
fixes $\bigwedge$:: *'form list* ⇒ *'form*
fixes $\Rightarrow$:: *'form* ⇒ *'form* ⇒ *'form*
fixes -,- ⊨ - :: *'prog* ⇒ (*'pos* × *'mem*) ⇒ *'form* ⇒ *bool*
fixes - ⊢ -:: *'prog* ⇒ *'form* ⇒ *bool*

Every structure having constants for the truth values $\lfloor T \rfloor$ and $\lfloor F \rfloor$, operators for conjunction $\bigwedge$ and implication $\Rightarrow$, judgments for validity ⊨ and provability ⊢ of formulas can be employed as a safety logic as long as it respects the requirements below. These requirements only concern the semantics (⊨) of the logical connectives. How formulas (*'form*) or their proofs (⊢) look like is left open to the instantiator.
To distinguish the symbols we use for the safety logic from our meta-logic (Isabelle/HOL), we frame the former with little corners $\lfloor\text{-}\rfloor$. Please do not confuse those with the floor symbols ⌊-⌋ we use for the *option* datatype. For better readability we sometimes use the infix $\barwedge$ instead of the prefix $\bigwedge$. For example $[A] \barwedge [B]$ or simply $A \barwedge B$ are just alternative writings for $\bigwedge [A,B]$. We also write the logical judgments in infix notation. With $\Pi,s \models f$ we express that for program Π the state s is a model of formula f. The notation $\Pi \vdash f$ means that for program Π the formula f is derivable in the safety logic.

Requirement 2.1 $\Pi,s \models \lfloor T \rfloor$

Requirement 2.2 $\neg\ (\Pi,s \models \lfloor F \rfloor)$

Requirement 2.3 $\Pi,s \models \bigwedge Fs = (\forall f \in set\ Fs.\ \Pi,s \models f)$

Requirement 2.4 $\Pi,s \models (f \Rightarrow f') \longrightarrow \Pi,s \models f \longrightarrow \Pi,s \models f'$

2.3 Safety Policy

The safety policy expresses what conditions we expect from programs to be safe. Safety conditions may vary from program position to program position. Our framework expects

safety conditions to be expressed via the safety logic.
That is, we represent the safety policy as a function *safeF*, that yields a safety formula for each program position. In locale *SafetyPolicy*, which extends *Semantics* and *SafetyLogic*, we declare this function *safeF* as a parameter.
What safety actually means only becomes clear after one instantiates *SafetyPolicy* with a concrete definition for *safeF*.

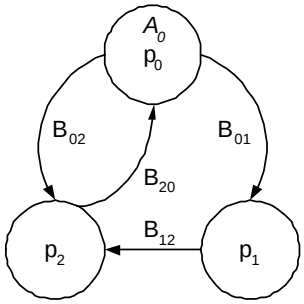
locale *SafetyPolicy* = *Semantics* + *SafetyLogic* +
fixes *safeF*:: *'prog* ⇒ *'pos* ⇒ *'form*

We call a program Π safe, i.e. *isSafe* Π, when all its reachable states (p,m) satisfy the safety formula at p, i.e. $\Pi,(p,m) \models$ *safeF* Π p.

isSafe::*'prog* ⇒ *bool*
isSafe Π = ∀ (p,m) ∈ *Reachables* Π. $\Pi,(p,m) \models$ *safeF* Π p

2.4 Annotated Control Flow Graphs

The VCG we define in this chapter views programs as annotated control flow graphs. Such graphs consist of positions, edges and annotations. Fig. 2.4 shows an example.



Example program Π_x

```
p0 :  if (x <= 0) {
p1 :  x = y + 2;  }
p2 :  x = x - 2;
      goto p0;
```

Figure 2.4: Annotated control flow graph.

We write *domC* Π for a program Π's code domain, which is a list of all its positions. Execution is expected to start at *ipc* Π, the initial position. In the example Π_x shown in Fig. 2.4, we have *domC* $\Pi_x = [p_0,p_1,p_2]$ and *ipc* $\Pi_x = p_0$. Note that the locale does not define this functionality, as it just declares *domC* and *ipc*. However, the requirements we state in further locales will constrain our parameter functions such that they behave as expected.

locale *CFG* = *SafetyPolicy* +
fixes *domC* :: *'prog* ⇒ *'pos list*
fixes *ipc* :: *'prog* ⇒ *'pos*
fixes *succsF* :: *'prog* ⇒ *'pos* ⇒ (*'pos* × *'form*) *list*
fixes *anF* :: *'prog* ⇒ *'pos* ⇒ *'form option*
fixes *wf* :: *'prog* ⇒ *bool*

Function *succsF* yields the edges of the control flow graph. Given a position p in a program Π the expression *succsF* Π p yields a list of pairs (p',B) where p' is a possible successor of p and B is the branch condition for the edge from p to p'. The branch condition B is a formula in the safety logic that describes the situations when p' is accessible from p. For example in Fig. 2.4 Π_x jumps from p_0 to p_1 under condition B_{01}, and from p_0 to p_2 under condition B_{02}. Hence, we have *succsF* Π_x p_0 = $[(p_1,B_{01}),(p_2,B_{02})]$.

With *anF* we access the annotations; *anF* Π p returns $\lfloor A \rfloor$ if position p in Π is annotated with A, otherwise *None*.
Sometimes it is more convenient to avoid the *option* type. With *aF* we introduce a variation of *anF* giving $\lfloor\!\lfloor \top \rfloor\!\rfloor$ for non-annotated positions and the annotation otherwise.

aF :: *'prog* ⇒ *'pos* ⇒ *'form*
aF Π p = (*case anF* Π p *of None* ⇒ $\lfloor\!\lfloor \top \rfloor\!\rfloor$ | $\lfloor A \rfloor$ ⇒ A)

For the list of all annotated positions we write *domA* Π.

domA :: *'prog* ⇒ *'pos list*
domA Π = [p∈ *domC* Π. *anF* Π $p \neq$ *None*]

Examples taken from Fig. 2.4 illustrate how these annotation functions work: *domA* Π_x = $[p_0]$, *anF* Π_x p_0 = $\lfloor A_0 \rfloor$, *anF* Π_x p_1 = *None*, *aF* Π_x p_0 = A_0 and *aF* Π_x p_1 = $\lfloor\!\lfloor \top \rfloor\!\rfloor$.

In the following we will often conjoin safety formulas with annotations, e.g. *safeF* Π p ◺ *aF* Π p. So, why don't we just use one function? The reason is that annotations are different for every program and are provided by the code producer. The safety formulas are provided by the consumer, who expresses with them the desired safety properties. Typically *safeF* is defined such that the formula *safeF* Π p only depends on the kind of instruction we find a position p in Π. The annotations are not local in that respect. In order to construct them the code producer typically has to take the entire arrangement of instructions in Π into account.

Finally, locale *CFG* introduces a wellformedness checker *wf* for programs. What wellformedness means is left open to the instantiator, except for some basic properties our framework demands. The most important is Requirement 2.5, which ensures that programs are sufficiently annotated. Our VCG requires every cycle in the control flow graph to have at least one annotation.

For the completeness proofs *wf* also has to check that *ipc* and *succsF* are closed under *domC*. Additional checks may be integrated to any instantiation of *wf*. Since *wf* is a premise of many requirements this may help to conduct the instantiation proofs. However, one should always keep in mind that *wf* is only meant to check simple properties. A code consumer must be able to run it efficiently. Please note that from now on we implicitly assume that we are dealing with wellformed programs.

Requirement 2.5 $wf\ \Pi \longrightarrow enoughAn\ \Pi$

The predicate *enoughAn* Π, which we define below, checks that every control flow cycle in Π has at least one annotation. To define cycles, we first introduce the concept of control flow paths.

$paths :: ('pos \Rightarrow ('pos \times 'form)\ list) \Rightarrow ('pos\ list)\ set$

$$\frac{(p', B) \in set\ (sc\ p)}{[p, p'] \in paths\ sc}\ \textsc{Pinit} \qquad \frac{l\ @\ [p] \in paths\ sc \qquad (p', B) \in set\ (sc\ p)}{l\ @\ [p]\ @\ [p'] \in paths\ sc}\ \textsc{Pstep}$$

A cycle is a control flow path, where the first and the last position coincide.

$isCycle :: 'prog \Rightarrow ('pos\ list) \Rightarrow bool$
$isCycle\ \Pi\ ps = ((hd\ ps = last\ ps) \wedge ps \in paths\ (succsF\ \Pi))$

Now, we are ready to define the predicate *enoughAn* from above.

$enoughAn :: 'prog \Rightarrow bool$
$enoughAn\ \Pi = (\forall ps.\ isCycle\ \Pi\ ps \longrightarrow (\exists p \in set\ ps.\ anF\ \Pi\ p \neq None))$

Sometimes we need further constraints on annotations. Above all they should be correct to be of any use for verification. We say a program Π is correctly annotated, i.e. *correctAn* Π, if all annotations hold at runtime.

$correctAn :: 'prog \Rightarrow bool$
$correctAn\ \Pi = \forall s \in Reachables\ \Pi.\ \Pi, s \models aF\ \Pi\ (fst\ s)$

2.5 Abstract Semantics

For verification purposes we work with an abstract semantics. It abstracts states with formulas and simulates transitions by manipulating these.

locale *AbsSem* = *CFG* +
fixes *initF*:: *'prog* ⇒ *'form*
fixes *wpF*:: *'prog* ⇒ *'pos* ⇒ *'pos* ⇒ *'form* ⇒ *'form*

Function *initF* Π can be seen as an abstract notion of *initS*. It abstractly models the set of initial states. Note that the letter F behind an identifier indicates that it belongs to the formal world, whereas S suggests the semantic side.

The weakest precondition operator *wpF* is the abstract counterpart of *effS*. The formula that *wpF* Π p p' Q yields is expected to characterise those states (p,m) that have successor states (p',m') satisfying Q. Note that we associate effects with edges, although instructions are sitting at nodes. This is because one instruction may behave differently. For example, consider *exceptional* versus *normal* execution. Having p and p' as arguments help *wpF* to figure out what effect must be simulated.

For our correctness and completeness proofs we require initial states to have the initial program counter determined by the control flow function *ipc*.

Requirement 2.6 *wf* Π ∧ Π,s ⊨ (*initF* Π) ⟶ *fst* s = *ipc* Π

2.6 Generic Verification Conditions

In this section we define a generic VCG, the core of our framework. We model the VCG as a function *vcg*::*'prog* ⇒ *'form* and construct verification conditions out of so called inductive safety formulas *isafeF* Π p, which we generate individually for each position p in a program Π. We call a state (p,m) *inductively safe* if it satisfies the inductive safety formula for p, i.e., Π,(p,m) ⊨ *isafeF* Π p.

We define *vcg* and *isafeF* in a locale called *VCG*, but need to use external functions *vcG* and *isafe* for this purpose. This is because locales in Isabelle/HOL do not (yet) support recursive definitions. Since the external functions *vcG* and *isafe* take all locale parameters as additional arguments their definitions are rather indigestible.

In addition, the recursive definition of *isafe* maintains a list of non-visited positions and terminates with ⌊F⌋ when this list becomes empty or a position is visited twice. For wellformed programs this is never the case, hence we are able to derive a more readable definition inside the locale. In this section, we only show the derived equations for *vcg* and *isafeF*. The definitions for *vcG* and *isafe* can be found in the appendix §A.2.1.

locale *VCG* = *AbsSem* +

isafeF :: *'prog* ⇒ *'pos* ⇒ *'form*
isafeF Π *p* = *isafe*(*domC* Π,Π,*anF*,*p*,⌞*F*⌟,⨅,⟹,*safeF*,*succsF*,*wpF*)

vcg :: *'prog* ⇒ *'form*
vcg Π = *vcG* ⨅ ⟹ ⌞*F*⌟ *ipc initF safeF succsF wpF domC domA anF* Π

In Fig. 2.5 we show the derived definition of *isafeF* Π *p*. The wellformedness check *wf* Π ensures that every cycle has at least one annotation; otherwise the recursion of *isafeF* would not terminate.

wf Π ⟶
isafeF Π *p* = (*if* (*p* ∈ *set* (*domC* Π))
then [*safeF* Π *p*] ⩓
(*case* (*anF* Π *p*)
of None ⇒ *map* (λ (*p'*,*B*). *B* ⟹ (*wpF* Π *p p'* (*isafeF* Π *p'*)))
(*succsF* Π *p*)
| ⌊*A*⌋ ⇒ [*A*])
else ⌞*F*⌟)

Figure 2.5: Construction of inductive safety formulas

When *p* lies outside the code domain *domC* Π we must never reach it at runtime. We express this formally by returning the unsatisfiable formula ⌞*F*⌟ in this case. For positions *p* within the code domain the inductive safety formula guarantees the safety formula *safeF* Π *p*. In addition, if there is an annotation *A* at *p*, we conjoin the safety formula with *A*. For example in Π_x from Fig. 2.4, we have the annotation A_0 at p_0. Hence, we obtain this inductive safety formula:

isafeF Π_x p_0 = [*safeF* Π_x p_0] ⩓ [A_0]

If *p* is not annotated, we take all successor positions *p'* together with their branch conditions *B* and recursively compute the inductive safety formulas *isafeF* Π *p'*. Using the *wpF* operator we construct a precondition *wpF* Π *p p'* (*isafeF* Π *p'*). If this precondition holds for a state (*p*,*m*) with some successor (*p'*,*m'*), then *isafeF* Π *p'* holds for (*p'*,*m'*). By constructing implications of the form *B* ⟹ (*wpF* Π *p p'* (*isafeF* Π *p'*)), we design the inductive safety formula *isafeF* Π *p* such that all states satisfying the branch condition *B* for a particular successor *p'* also have to satisfy the precondition above. These implications are constructed for all pairs (*p'*,*B*) we get from *succsF* Π *p*. For example the positions p_1 and p_2 are not annotated in Π_x. Below are their inductive

safety formulas, where *safeF*, *wpF*, branch conditions and annotations are not expanded. This can only be done after instantiating the parameter functions.

$$isafeF\ \Pi_x\ p_1 = [safeF\ \Pi_x\ p_1] \barwedge [B_{12} \barRightarrow wpF\ \Pi_x\ p_1\ p_2\ ([safeF\ \Pi_x\ p_2] \barwedge [B_{20} \barRightarrow wpF\ \Pi_x\ p_2\ p_0\ ([safeF\ \Pi_x\ p_0] \barwedge [A_0])])]$$

$$isafeF\ \Pi_x\ p_2 = [safeF\ \Pi_x\ p_2] \barwedge [B_{20} \barRightarrow wpF\ \Pi_x\ p_2\ p_0\ ([safeF\ \Pi_x\ p_0] \barwedge [A_0])]$$

Executing a program Π with an inductively safe state (p,m) produces a trace of inductively safe states until we reach the next annotated position p'. The state (p',m') in which we reach this position is safe and satisfies the annotation. After this state, the execution could become unsafe. However, this does not happen if all successor states of (p',m') are again inductively safe. This observation guides the construction of the verification condition *vcg* Π, which we show in Fig. 2.6.

$$\begin{aligned}
&wf\ \Pi \longrightarrow \\
&vcg\ \Pi = [initF\ \Pi \barRightarrow (isafeF\ \Pi\ (ipc\ \Pi))] \barwedge (\\
&\qquad map\ (\lambda p_a.\ \bigbarwedge (map\ (\lambda(p',B).\ ([isafeF\ \Pi\ p_a] \barwedge [B]) \barRightarrow \\
&\qquad\qquad\qquad\qquad wpF\ \Pi\ p_a\ p'\ (isafeF\ \Pi\ p'))\\
&\qquad\qquad\qquad (succsF\ \Pi\ p_a)))\\
&\qquad [p_a \in domC\ \Pi.\ anF\ \Pi\ p_a \neq None])
\end{aligned}$$

Figure 2.6: Verification Condition Generator

The verification condition *vcg* Π demands two things: First, all initial states must satisfy the first inductive safety formula $isafeF\ \Pi\ (ipc\ \Pi)$. Second, for every annotated position p_a the inductive safety formula $isafeF\ \Pi\ p_a$ and the branch condition B for all successors p' of p_a must guarantee $wpF\ \Pi\ p_a\ p'\ (isafeF\ \Pi\ p')$. This ensures that the transitions out of annotated positions lead to inductively safe successor states. Transitions out of non-annotated positions are automatically covered by the recursion in *isafeF*. Hence, it suffices to verify transitions out of annotated positions. For example *vcg* Π_x would have the following pattern:

$$\begin{aligned}
&vcg\ \Pi_x = [initF\ \Pi_x \barRightarrow (isafeF\ \Pi_x\ p_0)] \barwedge ([\\
&[([isafeF\ \Pi_x\ p_0] \barwedge [B_{01}]) \barRightarrow wpF\ \Pi_x\ p_0\ p_1\ (isafeF\ \Pi_x\ p_1)] \barwedge\\
&[([isafeF\ \Pi_x\ p_0] \barwedge [B_{02}]) \barRightarrow wpF\ \Pi_x\ p_0\ p_2\ (isafeF\ \Pi_x\ p_2)]])
\end{aligned}$$

The first conjunct expresses that initial states are inductively safe. Note that $ipc\ \Pi_x = p_0$. Since p_0 has two successors p_1 and p_2, which are accessible if B_{01} resp. B_{02} hold, we have two further conjuncts. One requires us to show that all states satisfying the inductive safety formula for p_0 and the branch condition B_{01} can only have successor states that satisfy the inductive safety formula for p_1. The other is analogous for p_2. Note that a verification condition contains as many conjuncts of the second form as

there are annotations in a program. One could reduce this number to a minimum by only annotating one position per loop. However, since the construction of inductive safety formulas analyses each path leading from one annotated position to the next, reducing the number of annotations increases the size of inductive safety formulas. So, it is advisable not only to annotate loops, but also positions where the control flow splits up and joins. This reduces the number of paths between annotations exponentially, while the number of annotations only increases linearly.

2.7 Correctness

Most important for a VCG is its correctness. If we can prove the verification condition for some program Π, then we want to know for sure that it is safe, i.e. *isSafe* Π.

Theorem 2.1 *(correctVCG)* Wellformed programs with provable verification condition are safe.
$wf\ \Pi \wedge \Pi \vdash vcg\ \Pi \longrightarrow isSafe\ \Pi$

Our proof of this theorem relies on various requirements on the parameter functions. If the instantiation can prove that its definitions for the parameters satisfy these requirements, then the correctness theorem automatically carries over to the instantiated VCG. Note that none of the requirements involve the safety policy *safeF*. This makes changing the safety policy very convenient. No proof needs to be adjusted. Locale *correctVCG* contains all the correctness requirements as well as the requirements it inherits from *VCG* and the other locales above it.

locale *correctVCG* = *VCG* +

Some requirements make use of the set *ReachablesAn* Π. It contains those states that are reachable by only traversing states that satisfy annotations.

ReachablesAn:: *'prog* $\Rightarrow$ (*'pos* $\times$ *'mem*) *set*
ReachablesAn Π = *ReachableFromInv* (*effS* Π) (*initS* Π) ({s. Π,$s \models aF\ \Pi$ (*fst* s)})

We say states in *ReachablesAn* Π are *anno-reachable* in order to distinguish them from states that are just reachable, i.e. *Reachables* Π. Note that only for correctly annotated programs both sets are equivalent.

Lemma 2.2 *(correctVCG) correctAn* $\Pi \longrightarrow$ *ReachablesAn* Π = *Reachables* Π

Many requirements only demand something for anno-reachable states. Note that this restriction makes it easier for the instantiator to prove the requirements. We come back to that below and in §6.5.

In the Requirements 2.7 and 2.8 we demand that the abstract semantics mimics the concrete one. Initial states must be covered by *initF*, and *wpF* must guarantee the postcondition in the successor state.

Requirement 2.7 $wf\ \Pi \wedge s \in initS\ \Pi \longrightarrow \Pi,s \models (initF\ \Pi)$

Requirement 2.8 $wf\ \Pi \wedge s \in (ReachablesAn\ \Pi) \wedge (s,s') \in (effS\ \Pi) \wedge$
$\Pi,s \models (wpF\ \Pi\ (fst\ s)\ (fst\ s')\ Q) \longrightarrow \Pi,s' \models Q$

Requirement 2.9 demands *succsF* to approximate the real control flow and that branch conditions are valid. In other words, *succsF* may guess successors, but must not miss any or give branch conditions that are too strong. Here the restriction to anno-reachable states is quite important. When one instantiates *succsF* to languages with procedures, one can stick annotations of call positions into branch conditions for edges leading back to this position (procedure return). As we explain in §6.5 this restores the call context and leads to modular verification conditions. We do not have to introduce a special procedure call/return treatment into our generic VCG.

Requirement 2.9 $wf\ \Pi \wedge s \in (ReachablesAn\ \Pi) \wedge (s,s') \in (effS\ \Pi) \longrightarrow$
$(\exists B.\ (fst\ s',B) \in set\ (succsF\ \Pi\ (fst\ s)) \wedge \Pi,s \models B)$

Finally, Requirement 2.10 demands the safety logic to be correct. Derivable formulas must hold for all anno-reachable states. Note that this is a weaker form of correctness than usually stated in logic textbooks. Normally, logical correctness demands provable formulas to be tautologies. For our purpose of proving safety and correctness of annotations it suffices if the verification condition holds for all anno-reachable states. The additional premise, which restricts s to be anno-reachable, can simplify the proof of this requirement. In [96] we defined $\Pi \vdash f$ such that it holds for all invariant formulas f, not just for tautologies. Having the additional premise was crucial there to prove Requirement 2.10.

Requirement 2.10 $wf\ \Pi \wedge \Pi \vdash f \wedge s \in (ReachablesAn\ \Pi) \longrightarrow \Pi,s \models f$

Correctness Proof

Proof (Theorem 2.1) We obtain our correctness Theorem 2.1 from Lemma 2.3 below. In case of a provable verification condition it states that all reachable state are inductively safe. Since inductively safe states are also safe, we can establish program safety, i.e. *isSafe* Π. □
Theorem 2.2 also follows from Lemma 2.3 and is important, because annotations usually come from the same untrusted source as the code. Indirectly Theorem 2.2 says that incorrect annotations lead to unprovable verification conditions.

Theorem 2.2 *(correctVCG)* Wellformed programs with provable verification condition are correctly annotated.

$wf\ \Pi \wedge \Pi \vdash vcg\ \Pi \longrightarrow correctAn\ \Pi$

Proof All anno-reachable states satisfy the annotations. Lemma 2.3 guarantees that all reachable states are anno-reachable, hence *correctAn* Π. □

All that remains is to state and prove Lemma 2.3. This shows why the requirements are in the form presented above and how they fit together.

Lemma 2.3 *(correctVCG)* In wellformed programs with provable verification conditions all reachable states are inductively safe and anno-reachable.

$wf\ \Pi \wedge \Pi \vdash vcg\ \Pi \wedge s \in (Reachables\ \Pi) \longrightarrow (\Pi, s \models isafeF\ \Pi\ (fst\ s) \wedge s \in ReachablesAn\ \Pi)$

Proof We prove Lemma 2.3 by induction on *Reachables* Π. Since *Reachables* Π is defined via the set *ReachFromIns*, we can use the induction rule Isabelle/HOL automatically provides for inductive definitions. In the base case we have $s_0 \in initS\ \Pi$. From this we get $s_0 \in ReachablesAn\ \Pi$ and derive $\Pi, s_0 \models initF\ \Pi$ using Requirement 2.7. Using Requirement 2.6 and the initial conjunct of the verification condition, which holds for s_0 because of Requirement 2.10, we arrive at $\Pi, s_0 \models isafeF\ \Pi\ (ipc\ \Pi)$, which finishes the base goal.
In the induction case, we can assume $(s, s') \in (effS\ \Pi)$, $\Pi, s \models isafeF\ \Pi\ (fst\ s)$ and $s \in ReachablesAn\ \Pi$. Our goals are $\Pi, s' \models isafeF\ \Pi$ and $s' \in ReachablesAn\ \Pi$. First we make a case distinction on $anF\ \Pi\ (fst\ s)$.
If there is some annotation A, i.e. $anF\ \Pi\ (fst\ s) = \lfloor A \rfloor$, we use the verification condition, which holds for s because of Requirement 2.10 and our assumptions. In the annotation case $fst\ s$ belongs to $domA\ \Pi$ and because of $(s, s') \in effS\ \Pi$ and Requirement 2.9 we find some branch condition B, such that $(fst\ s', B) \in (succsF\ \Pi\ (fst\ s))$ and $\Pi, s \models B$. This means the verification condition has a conjunct of the form $(safeF\ \Pi\ (fst\ s) \mathbin{\leftthreetimes} A \mathbin{\leftthreetimes} B) \mathbin{\Rightarrow\!\!\!\!\!\!\rightthreetimes} wpF\ \Pi\ (fst\ s)\ (fst\ s')\ (isafeF\ \Pi\ (fst\ s'))$, which holds also for s because of

Requirement 2.3 and the fact that the entire verification condition holds for s. From $\Pi,s \models \mathit{isafeF}\ \Pi\ (\mathit{fst}\ s)$ we get $\Pi,s \models \mathit{safeF}\ \Pi\ (\mathit{fst}\ s) \mathbin{\triangle} A$. Hence all the conditions of the left hand side of this implication formula hold. Using Requirement 2.4, we get $\Pi,s \models \mathit{wpF}\ \Pi\ (\mathit{fst}\ s)\ (\mathit{fst}\ s')\ (\mathit{isafeF}\ \Pi\ (\mathit{fst}\ s')$. Now, we can use Requirement 2.8 and get our goal $\Pi,s \models (\mathit{isafeF}\ \Pi\ (\mathit{fst}\ s'))$. For the second goal, we first establish $\Pi,s' \models \mathit{aF}\ \Pi\ (\mathit{fst}\ s')$ and use the R rule for *ReachablesAn*. To show $\Pi,s' \models \mathit{aF}\ \Pi\ (\mathit{fst}\ s')$ we make a case distinction on $\mathit{anF}\ \Pi\ (\mathit{fst}\ s')$. In the *None* case $\mathit{aF}\ \Pi\ (\mathit{fst}\ s')$ yields $\underline{\top}$, which holds because of Requirement 2.1. In the $\mathit{anF}\ \Pi\ (\mathit{fst}\ s') = \lfloor A' \rfloor$ case, we get $\Pi,s \models A'$ from $\Pi,s' \models \mathit{isafeF}\ \Pi\ (\mathit{fst}\ s')$, the first goal shown before. This concludes the case $\mathit{anF}\ \Pi\ (\mathit{fst}\ s) = \lfloor A \rfloor$.
In the other case, that is $\mathit{anF}\ \Pi\ (\mathit{fst}\ s) = \mathit{None}$, we do not need the verification condition. In this case we get the important fact from the definition of *isafeF* and Requirement 2.9. We get $\Pi,s \models \mathit{safeF}\ \Pi\ (\mathit{fst}\ s) \mathbin{\triangle} B \mathbin{\Rightarrow\!\!\!\!\!\!\sqsubset} \mathit{wpF}\ \Pi\ (\mathit{fst}\ s)\ (\mathit{fst}\ s')\ (\mathit{isafeF}\ \Pi\ (\mathit{fst}\ s')$ from $\Pi,s \models \mathit{isafeF}\ \Pi\ (\mathit{fst}\ s)$. Now we can use Requirements 2.8, 2.3 and 2.4 like before and arrive at $\Pi,s' \models \mathit{isafeF}\ \Pi\ (\mathit{fst}\ s')$. From that, we can show the goal $s' \in \mathit{ReachablesAn}\ \Pi$ like before. □

2.8 Completeness

Apart from correctness we have also proven that our VCG is relatively complete [34]. Before we explain what we mean by that, let us have a look at Hoare Logic and what completeness means there.
In Hoare Logic one specifies a program Π's input/output behaviour in form of a triple $\{P\}\ \Pi\ \{Q\}$. A Hoare triple is valid, i.e. $\models \{P\}\ \Pi\ \{Q\}$, if Π, when started on a state satisfying P, only terminates in states satisfying Q. The Hoare Logic itself consists of rules on how to derive such triples $\vdash \{P\}\ \Pi\ \{Q\}$. A Hoare Logic is complete, if every valid triple is derivable. For most Hoare Logics, this is not the case as they use incomplete logics for the assertions and some rules, such as the consequence rule, demand to prove side conditions in the assertion logic.
Hence, the best one can usually have is a Hoare Logic that is relatively complete. In case of relative completeness a valid Hoare triple has a derivation where all the side conditions are valid assertions, and the assertion logic is expressive. The latter means that we can encode all assertions a derivation requires internally, such as loop invariants.
Our aim is now to adapt this notion of completeness to our VCG, which essentially is just a machine applying Hoare-like rules internally and emitting all the side conditions as one verification condition. However, there are also things that are different. First, we are primarily interested in safety and not so much in functional correctness. For us not only initial and final states count, but also intermediate ones. Second, we do not care

about termination, since also non-terminating programs can be safe.
Hence, in our case a specification not only involves annotations at initial and final control flow positions, but also ones in between. Nevertheless we can interpret our annotations as embedded Hoare triples. Every path between two control flow positions p and q can be interpreted as an embedded program. If p is annotated with P and q is annotated with Q (if not, take $\lfloor\top\rfloor$ for Q) we can build the triple $\{P\}\ p \ldots q\ \{Q\}$ and view it as an embedded Hoare triple.
For example in Fig. 2.4 we find the embedded Hoare triple $\{A_0\}\ p_0,p_1,p_2,p_0\ \{A_0\}$. In addition there is $\{A_0\}\ p_0,p_1,p_2,p_0,p_2,p_0\ \{A_0\}$, or $\{A_0\}\ p_0,p_1\ \{\lfloor\top\rfloor\}$ and so on. We call an embedded Hoare triple $\{P\}\ ps\ \{Q\}$ valid, if all execution traces in *effS* that strictly follow the path ps end with a state satisfying Q provided they start with one satisfying P.
Now, if we also add the safety formulas of the initial and final path position to our triple, we arrive at what we call a *safety triple*. For example if S_i abbreviates *safeF* Π_x p_i then Fig. 2.4 contains the safety triples $\{A_0 \between S_0\}\ p_0,p_1,p_2,p_0\ \{A_0 \between S_0\}$ or $\{A_0 \between S_0\}\ p_0,p_1\ \{\lfloor\top\rfloor \between S_1\}$ and many others. We say a program Π is strongly annotated, i.e. *strongAn* Π, if all embedded safety triples are valid.
To define this formally, we do not have to formalise the concept of a safety triple, but can instead use our reachability predicates. We introduce *Starters* Π, which contains two kinds of states. It contains all states satisfying *initF* Π. In addition it contains all safe states s that have an annotation and satisfy it. The latter means that *fst s* is annotated, i.e. *anF* Π (*fst s*) $= \lfloor A \rfloor$, and this annotation holds for s, i.e. $\Pi,s \models A$.

locale *completeVCG* = *VCG* +

Starters:: *'prog* ⇒ (*'pos* × *'mem*) *set*
Starters Π = {*s*. Π,*s* ⊨ *initF* Π ∨
(∃ *A*. *anF* Π (*fst s*) = ⌊*A*⌋ ∧ Π,*s* ⊨ *A* ∧ Π,*s* ⊨ *safeF* Π (*fst s*))}

Instead of starting programs on initial states only, we now consider execution from any state in *Starters* Π. We only have to consider executions that stay within the control flow graph and where all transitions satisfy a branch condition. The transition relation $effS_B$ constrains *effS* in that respect.

$effS_B$ :: *'prog* ⇒ ((*'pos* × *'mem*) × (*'pos* × *'mem*)) *set*
$effS_B$ Π = *effS* Π ∩ {((*p*,*m*),(*p'*,*m'*)). ∃ *B*. (*p'*,*B*) ∈ *set* (*succsF* Π *p*) ∧ Π,(*p*,*m*) ⊨ *B*}

A program is strongly annotated, if all states $effS_B$ Π can reach from a state in *Starters* Π are safe and satisfy their annotation (in case there is one).

strongAn:: *'prog* ⇒ *bool*
strongAn Π = (∀ *s* ∈ *ReachableFrom* ($effS_B$ Π) (*Starters* Π).
Π,*s* ⊨ *aF* Π (*fst s*) ∧ Π,*s* ⊨ *safeF* Π (*fst s*))

Note that *strongAn* contains severe restrictions on the annotations and safety formulas. Both together must constrain each state such that all further states we can reach by executing Π along a control flow path also satisfy their annotations and safety formulas.

In the example from Fig. 2.4 we could annotate p_0 with a formula stating that x equals 0, e.g. $A_0 = (x = 0)$. Then if we assume that in all initial states x and y are initialised to 0, the program Π_x would be correctly annotated, i.e. *correctAn* Π_x. However, it would not be strongly annotated, as *Starters* Π_x not only contains initial states, but also all states satisfying A_0. The state $(p_0,(x{:}0,y{:}1))$ would be in *Starters* Π_x as A_0 does not restrict y. Now, if we start executing Π_x with this state and follow the path p_0,p_1,p_2,p_0, we end up with the state $(p_0,(x{:}1,y{:}1))$, which violates A_0. If we strengthen A_0 to $A_0 = (x = 0 \wedge y = 0)$ then Π_x is strongly annotated, provided it respects the safety policy.

The predicate *strongAn* Π has close similarities with the concept of inductive invariants as defined in [90]. For a transition system (R,F) a set of states I is an invariant, if it covers all reachable states, i.e. $\forall\, s \in \mathit{Reachables}\ R\ F.\ s \in I$. An invariant I is an *inductive invariant* , if we also have: $s \in I \wedge (s,s') \in R \longrightarrow s' \in I$. Now, if we assume that every position is annotated, and consider the set of states $F = \{s.\ \Pi,s \models \mathit{initF}\ \Pi\}$ and $AS = \{s.\ \Pi,s \models \mathit{aF}\ \Pi\ (\mathit{fst}\ s) \wedge \Pi,s \models \mathit{safeF}\ \Pi\ (\mathit{fst}\ s)\}$ as well as the transition relation $R = (\mathit{effS}_B\ \Pi)$. Then *strongAn* Π says that $F \cup AS$ is an inductive invariant for $(R,F \cup AS)$. Note that we can prove the theorems below also by taking *effS* instead of effS_B, but this just makes *strongAn* Π stronger than necessary.

Now, we are able to state our relative completeness theorem for the VCG. It says that strongly annotated programs have valid verification conditions.

Theorem 2.3 *(completeVCG)* For wellformed and strongly annotated programs the verification condition is a tautology.

$\mathit{wf}\ \Pi \wedge \mathit{strongAn}\ \Pi \longrightarrow (\forall\ s.\ \Pi,s \models \mathit{vcg}\ \Pi)$

Note that we are talking about completeness of the VCG, and not of the safety logic. In the latter case we would have $\Pi \vdash \mathit{vcg}\ \Pi$ in the conclusion. Since the safety logic we instantiate later contains non-linear arithmetics and is thus naturally incomplete, our framework makes no requirement on the completeness of the safety logic, i.e. $(\forall\, s.\ \Pi,s \models f) \longrightarrow \Pi \vdash f$. This means we might end up with a valid verification condition, but cannot construct a proof for it. However, in this case it is the safety logic we have to blame, not the VCG, which might give us proof obligations that are overly restrictive.

Before we come to the proof of theorem 2.3 let us have a look on the requirements for the parameter functions.

In Requirement 2.11 we demand that *succsF* gives us a precise control flow graph.

Whenever a branch condition for an edge holds a transition along this edge must be possible. This means, branch conditions guarantee progress.

Requirement 2.11 $wf\ \Pi\ \wedge \Pi,(p,m) \models B \wedge\ (p',B) \in set\ (succsF\ \Pi\ p'')$
$\longrightarrow (p{=}p'' \wedge (\exists\, m'.\ ((p,m),(p',m')) \in (effS\ \Pi)))$

Requirements 2.12 and 2.13 demand that *ipc* and *succsF* only yield results within the code domain *domC*. Note that these requirements together with requirement 2.11 enforce control flow safety, no matter how *safeF* becomes instantiated. Control flow safety is a fundamental basis for static code verification. It ensures that during execution the program counter always stays within the inspected code.

Requirement 2.12 $wf\ \Pi \longrightarrow ipc\ \Pi \in set\ (domC\ \Pi)$

Requirement 2.13
$wf\ \Pi \wedge (p',B) \in set\ (succsF\ \Pi\ p) \longrightarrow (p \in set\ (domC\ \Pi) \wedge p' \in set\ (domC\ \Pi))$

Requirement 2.14 states an important property for the abstract semantics. The *wpF* function must now compute weakest preconditions. In other words, the formulas it yields must not be too strong. Whenever a successor state satisfies the postcondition Q the weakest precondition must hold for the current state.

Requirement 2.14
$wf\ \Pi \wedge ((p,m),(p',m')) \in (effS_B\ \Pi) \wedge \Pi,(p',m') \models Q \longrightarrow \Pi,(p,m) \models wpF\ \Pi\ p\ p'\ Q$

Finally, the safety logic must guarantee the implication introduction rule. For the correctness proof this was not necessary, hence we did not include this property in the general assumptions on the safety logic.

Requirement 2.15 $wf\ \Pi \wedge (\Pi,s \models f \longrightarrow \Pi,s \models f') \longrightarrow \Pi,s \models f \Rightarrow f'$

Completeness Proof

The proof of Theorem 2.3 is based on Lemma 2.4, whose premise partitions the code domain into a list of visited positions V and spare positions S.

Lemma 2.4 *(completeVCG)* In wellformed and strongly annotated programs, all states s reachable from *Starters* Π are inductively safe, provided we can find a control flow path V of non-annotated positions leading to position *fst s* inside the code domain.

$wf\ \Pi \wedge strongAn\ \Pi \longrightarrow (\forall\, s \in ReachableFrom\ (effS_B\ \Pi)\ (Starters\ \Pi).$
$((\exists\, V.\ (\forall\, p \in set\ V.\ anF\ \Pi\ p = None) \wedge set\ V \cup set\ S = set\ (domC\ \Pi)$
$\wedge\ set\ V \cap set\ S = \{\} \wedge (V \neq [] \longrightarrow V@[fst\ s] \in (paths\ (succsF\ \Pi))))$
$\longrightarrow ((fst\ s \in set\ (domC\ \Pi)) \longrightarrow (\Pi,s \models isafeF\ \Pi\ (fst\ s)))))$

Proof Lemma 2.4 contains a sophisticated constraint on a subset S of the code domain. It plays a vital role for our induction on S. If we take apart the constraints for S and V and s, we can assume:
(1) no position in V is annotated.
(2) V and S partition the code domain $domC\ \Pi$.
(3) V is empty or can be augmented by $fst\ s$ to a control flow path.
(4) $fst\ s$ lies in $domC\ \Pi$.
(5) s is reachable from $Starters\ \Pi$.
Our goal is to show that s is inductively safe. We prove this by induction on the length of S, which will decrease with every recursion of $isafeF$.
In the base case we have $S = []$. From (2) we get $V = domC\ \Pi$. This means V cannot be empty, because Requirement 2.12 says that it contains at least $ipc\ \Pi$. Therefore (3) says we have a control flow path containing all positions and $fst\ s$. Because of (4) we have that $fst\ s$ must already be in V and thus occurs twice in the control flow path. Since (1) says that there are no annotations in V, we have a cycle with no annotation. This contradicts $wf\ \Pi$ and Requirement 2.5, which demands cycles to have at least one annotation.
In the induction case, we have the hypothesis that any s' is inductively safe, provided we can find some S' with a smaller length than S and some V' such that (1), (2), (3), (4), (5) hold for s, S and V replaced by s', S' and V'. We refer to these modified conditions as (1'), (2'), (3'), (4') and (5') and continue with case distinctions. First, a case distinction on $anF\ \Pi\ (fst\ s)$. Assume there is some annotation A, i.e. $anF\ \Pi\ (fst\ s) = \lfloor A \rfloor$. We know from $strongAn\ \Pi$ and (5) that s is safe and satisfies A. Because s is annotated $isafeF\ \Pi\ (fst\ s)$ demands exactly those two conditions.
In case there is no annotation at $fst\ s$, i.e. $anF\ \Pi\ (fst\ s) = None$, we make another case distinction on $fst\ s \in set\ S$.
Assume $fst\ s \in set\ S$ holds. We unfold our goal formula $isafeF\ \Pi\ (fst\ s)$ with the definition of $isafeF$ shown in Fig. 2.5. Because there is no annotation and (4) holds, we end up in the recursive branch. In case the control flow graph does not have any successor for s, we are done, as the goal formula becomes an empty conjunction. Otherwise, we take some successor p' together with its branch condition B, i.e. $(p',B) \in set\ (succsF\ \Pi\ (fst\ s))$. Our new goal is to show that B implies the weakest precondition for the inductive safety formula at p', i.e $\Pi,s \models B \Rightarrow wpF\ \Pi\ (fst\ s)\ p'\ (isafeF\ \Pi\ p')$. Using Requirement 2.15 we can assume that the branch condition holds,i.e. $\Pi,s \models B$, and have

to show the weakest precondition only, i.e. $\Pi,s \models wpF\ \Pi\ (fst\ s)\ p'\ (isafeF\ \Pi\ p')$. All premises of Requirement 2.11 are met. Hence, we conclude that there is some m', such that $(s,(p',m')) \in effS_B\ \Pi$. Now, we apply Requirement 2.14, which solves our goal, if we can show that the postcondition holds for the successor state, i.e. $\Pi,(p',m') \models isafeF\ \Pi\ p'$. Here the induction hypothesis comes into play. Note that we are still in the case $fst\ s \in set\ S$. Hence, we can instantiate S' with $[q \in S.\ q \neq fst\ s]$, which is clearly shorter than S. For V' we take $V@[fst\ s]$ and for s' we take (p',m'). In this instantiation the induction hypothesis guarantees our goal, provided we can show (1'), (2'), (3'), (4') and (5'). Since $fst\ s$ is not annotated, we obtain (1') from (1). From (2) and the fact that we just moved one element from one list to the other, S' and V' are still a partition of $domC\ \Pi$. Hence, we have (2'). Since, p' comes from $succsF\ \Pi\ (fst\ s)$, we can augment our former control flow path to $V@[fst\ s,p']$, which gives us (3'). To establish (4') we use Requirement 2.13. Finally, we have (5'), because we can apply the R rule to (5) and $(s,(p',m')) \in (effS_B\ \Pi)$. This finishes the case $fst\ s \in set\ S$.
When s is not in S, i.e. $fst\ s \notin set\ S$, it must lie in V, i.e. $fst\ s \in set\ V$, because it belongs to $domC\ \Pi$ (4), which is partitioned by S and V (2). Hence, V is not empty and we get from (3) that $V@[fst\ s]$ is a control flow path. Since, $fst\ s$ occurs twice in this path and no position is annotated, we have again a cycle with no annotation. This contradicts $wf\ \Pi$ and Requirement 2.5. □

From Lemma 2.4 we can easily derive Lemma 2.5.

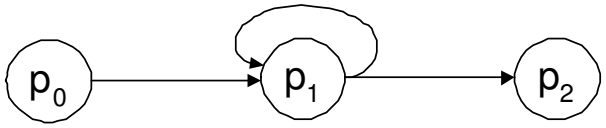
Lemma 2.5 *(completeVCG)* For wellformed and strongly annotated programs every state reachable from *Starters* Π is inductively safe, provided its program counter lies within the code domain.

$wf\ \Pi \wedge strongAn\ \Pi \longrightarrow (\forall\ s \in ReachableFrom\ (effS_B\ \Pi)\ (Starters\ \Pi).$
$(fst\ s \in set\ (domC\ \Pi)) \longrightarrow \Pi,s \models isafeF\ \Pi\ (fst\ s))$

Proof Take Lemma 2.4 and instantiate S with $domC\ \Pi$ and V with []. All the conditions for S and V hold trivially. □

Now, let us see how this proves Theorem 2.3.

Proof (Theorem 2.3) For the initial conjunct of the verification condition, we have to prove $\Pi,s \models (initF\ \Pi) \rightrightarrows (isafeF\ \Pi\ (ipc\ \Pi))$. Requirement 2.15 allows us to assume $\Pi,s \models (initF\ \Pi)$ and reduces the goal to $\Pi,s \models isafeF\ \Pi\ (ipc\ \Pi)$. With Requirement 2.6 we can transform this to $\Pi,s \models isafeF\ \Pi\ (fst\ s)$. Because of Requirement 2.12 we get that $ipc\ \Pi$ or $fst\ s$ belongs to $domC\ \Pi$. The initial formula $initF$ holds for s, which is thus in $Starters\ \Pi$. Rule F guarantees that s also lies in $ReachableFrom\ (effS_B\ \Pi)\ (Starters\ \Pi)$. This gives us all conditions to apply Lemma 2.5 to finish this goal.
To show the other conjuncts of the verification condition, we pick some annotated position p, i.e. $anF\ \Pi\ p = \lfloor A \rfloor$, with some successor p' under branch condition B, i.e. (p',B)



$specF\ \Pi\ p_0 = \lfloor initF\ \Pi \rfloor \quad specF\ \Pi\ p_1 = None \quad specF\ \Pi\ p_2 = \lfloor Q \rfloor$

Figure 2.7: Functional Specification

$\in succsF\ \Pi\ p$. This leaves us with the task to prove $\Pi,s \models (isafeF\ \Pi\ p \mathbin{\triangle} B) \mathrel{\rightrightarrows} wpF\ \Pi\ p\ p'\ (isafeF\ \Pi\ p')$. Again, Requirement 2.15 allows us to get rid of $\rightrightarrows$ in the goal. Under the assumptions $\Pi,s \models isafeF\ \Pi\ p$ and $\Pi,s \models B$, we have to show $wpF\ \Pi\ p\ p'$ $(isafeF\ \Pi\ p')$. From the first assumption, we get that s satisfies the annotation at p, i.e. $\Pi,s \models aF\ \Pi\ p$. Like in the proof for Lemma 2.4, we can use Requirements 2.11 and 2.14 to reduce our goal to $\Pi,(p',m') \models (isafeF\ \Pi\ p')$ for some m' such that $(s,(p',m'))$ $\in (effS_B\ \Pi)$. From Requirement 2.11, we also get $fst\ s = p$, which means that s satisfies the annotation at $fst\ s$ and is thus in $Starters\ \Pi$. Because s can make a transition to (p',m') we can apply rule R to obtain $s \in ReachableFrom\ (effS_B\ \Pi)\ (Starters\ \Pi)$. From Requirement 2.13 we obtain $p' \in set\ (domC\ \Pi)$. Now, we can instantiate our Lemma 2.5 with s replaced by (p',m') to finish the proof. □

Expressiveness

Theorem 2.3 states that we obtain valid verification conditions for strongly annotated programs. The question is: *Can we always find strong annotations?*
Assume we have a program Π and a functional specification for it in form of a Hoare triple $\{initF\ \Pi\}\ \Pi\ \{Q\}$. In our framework we can express this specification with a function *specF* that assigns $initF\ \Pi$ to the initial position, Q to all final positions and nothing to all internal positions. If Π only has three positions we have the situation depicted in Fig. 2.7. Formally, a specification is a function *specF* that assigns formulas to initial and final positions.

locale *Expressiveness* = *completeVCG* +

specF:: *'program* ⇒ *'pos* ⇒ *'form*

For the initial position *ipc* Π the specification must coincide with $initF\ \Pi$.

Requirement 2.16 $specF\ \Pi\ (ipc\ \Pi) = \lfloor initF\ \Pi \rfloor$

According to requirement 2.17 a specification must also provide formulas for all final, but not for internal positions. Final positions are those in the set *finals* Π , which contains

all positions p where the control flow function *succsF* Π p does not have an outgoing edge. If a position is neither initial nor final we call it internal.

finals :: *'program* $\Rightarrow$ *'pos set*
finals $\Pi = \{p.\ p \in set\ (domC\ \Pi) \wedge \neg\ (\exists p'\ B.\ (p',B) \in set\ (succsF\ \Pi\ p))\}$

Requirement 2.17 $\forall p.\ p \in \{ipc\ \Pi\} \cup (finals\ \Pi) = (specF\ \Pi\ p \neq None)$

Now, if we annotate a program according to its specification, i.e. $anF = specF$, then *strongAn* Π holds, if the specification is valid, i.e. $\models \{initF\ \Pi\}\ \Pi\ \{Q\}$, and the program can only reach safe states from $\{s.\ \Pi, s \models initF\ \Pi\}$. However, as soon as we start adding annotations to internal positions *strongAn* Π may break, because *Starters* Π additionally receives the states satisfying the new annotation and this annotation may not be strong enough to guarantee all further ones.
Hoare Logic also uses internal annotations, but these only occur in the derivation of a Hoare triple $\vdash \{P\}\ \Pi\ \{Q\}$. It is part of the completeness proof to show that provided a triple is valid, i.e. $\models \{P\}\ \Pi\ \{Q\}$, one can always construct the required internal annotations. This is what the literature addresses with the notion *expressiveness*.
According to Winskel [99] an assertion logic is expressive when it can encode weakest preconditions. In requirements 2.8 and 2.14 we demand that *wpF* yields preconditions that are correct and weak enough. So can we conclude from that that our safety logic is expressive? We can not, because our *wpF* operator only approximates the effect of single control flow edges, but Winskel is talking about weakest preconditions for structured programs.
Since Winskel's proof on expressiveness is by induction on the program structure porting it to our framework becomes difficult. In addition Winskel employs an assertion logic with quantifiers and arithmetic expressions. These are needed to construct invariants for *While* loops. The idea is to express program execution with a formula on natural numbers. States and finite sequences of states can be encoded as natural numbers. This does not work for infinite sequences, hence Winskel's completeness proof requires programs to terminate.
Requiring termination and extending our safety logic to first order arithmetics would clearly reduce the generality of our framework. Therefore, we set out for a different approach to completeness. The question is, under what conditions can we construct all the required internal annotations in a propositional style?
As we will show, this is possible, when a program Π has a finite diameter. The *diameter* of a transition relation $effS_B\ \Pi$ is the highest distance between any two states. This notion is heavily used in bounded model checking [20] and gives analogous completeness results there. In reality programs run on computers with limited memory and thus always have a finite diameter. To define diameters formally, we introduce the set *ReachableFromIn* $R\ F\ k$, which contains all states R reaches from F in exactly k steps.

ReachableFromIn :: $('a \times 'a)\ set \Rightarrow 'a\ set \Rightarrow nat \Rightarrow 'a\ set$

$$\frac{a \in F}{a \in \mathit{ReachableFromIn}\ R\ F\ 0}\ \mathrm{F} \qquad \frac{a \in \mathit{ReachableFromIn}\ R\ F\ k \qquad (a,\ b) \in R}{b \in \mathit{ReachableFromIn}\ R\ F\ (k\ +\ 1)}\ \mathrm{R}$$

The diameter of $\mathit{effS}_B\ \Pi$ is finite, if we can find a bound d for it. Whenever $\mathit{effS}_B\ \Pi$ reaches a state s' from another state s it can also reach it within d steps.

diameter :: $('s \times 's)\ set \Rightarrow nat \Rightarrow bool$
$\mathit{diameter}\ R\ d = \forall\, s\ s'.\ s' \in \mathit{ReachableFrom}\ R\ \{s\} \longrightarrow (\exists\, r{\leq}d.\ s' \in \mathit{ReachableFromIn}\ R\ \{s\}\ r)$

When we have an upper bound d for the diameter, we can construct internal annotations for a given functional specification *specF*. For this purpose we introduce the function *ispecF*, which computes internal safety specifications.

ispecF :: *'program* $\Rightarrow$ *'pos* $\Rightarrow$ *nat* $\Rightarrow$ *'form*

$\mathit{ispecF}\ \Pi\ p\ k = (\mathsf{case}\ \mathit{specF}\ \Pi\ p\ \mathsf{of}\ \lfloor Q \rfloor \Rightarrow Q$
$\mid \mathit{None} \Rightarrow (\mathsf{case}\ k\ \mathsf{of}\ 0 \Rightarrow \lfloor\!\lceil \bot \rceil\!\rfloor$
$\quad \mid \mathit{Suc}\ k' \Rightarrow \bigsqcap\!\!\!\bigsqcup\ (\mathit{map}\ (\lambda(p', B).\ B \sqsupset\!\!\!\sqsubset (\mathit{wpF}\ \Pi\ p\ p'\ (\bigsqcap\!\!\!\bigsqcup\ [\mathit{safeF}\ \Pi\ p',\ \mathit{ispecF}\ \Pi\ p'\ k'])))$
$\qquad\qquad (\mathit{succsF}\ \Pi\ p)))$

The construction is very similar to *isafeF*, except that *ispecF* can unroll cycles multiple times if d is large enough. When p is the initial or a final position, then *ispecF* yields *specF* Π p. If p is an internal position, that is *specF* Π p = *None*, then *ispecF* constructs a formula that ensures that all states that can be reached from p within the next d steps are safe and satisfy their (internal) specification. This implies that if we start executing a program Π on a state (p,m) that satisfies *ispecF* Π p, then every final state (p',m') we reach within d steps satisfies the postcondition *specF* Π p.

This means, we can use *ispecF* to fully annotate a given program that only has annotations at initial and final positions. The next theorem shows, that the annotations we construct in this way are strong.

Theorem 2.4 *(Expressiveness)* Wellformed programs with a bounded diameter that are safe and satisfy a given specification *specF* can be strongly annotated with *ispecF*.

$\mathit{wf}\ \Pi \wedge \mathit{diameter}\ (\mathit{effS}_B\ \Pi)\ d \wedge\ (\forall\, s \in \mathit{ReachableFrom}\ (\mathit{effS}_B\ \Pi)\ \{s.\ \Pi, s \models \mathit{initF}\ \Pi\}.$
$\Pi, s \models \mathit{safeF}\ \Pi\ (\mathit{fst}\ s) \wedge (\forall\, Q.\ \mathit{specF}\ \Pi\ (\mathit{fst}\ s) = \lfloor Q \rfloor \longrightarrow \Pi, s \models Q))$
$\wedge\ (\forall\, p.\ \mathit{anF}\ \Pi\ p = \lfloor \mathit{ispecF}\ \Pi\ p\ d \rfloor) \longrightarrow \mathit{strongAn}\ \Pi$

The proof relies on two further requirements. We forbid transitions back to the initial position, because *ispecF* Π (*ipc* Π) does not guarantee anything for follow up positions.

Requirement 2.18 *wf* $\Pi \longrightarrow (\forall\, p\ p'\ B.\ (p',B) \in set\ (succsF\ \Pi\ p) \longrightarrow p' \neq ipc\ \Pi)$

Finally, we assume that *wpF* is correct for all state transitions in $effS_B$. Note that requirement 2.19 turns the implication in requirement 2.14 around. This requirement is harder to fulfil than requirement 2.8, because we now do not have the constraint that (p,m) is a reachable state.

Requirement 2.19
wf $\Pi \wedge ((p,m),(p',m')) \in (effS_B\ \Pi) \wedge \Pi,(p,m) \models wpF\ \Pi\ p\ p'\ Q \longrightarrow\ \Pi,(p',m') \models Q$

Proof (Theorem 2.4) By unfolding *strongAn* Π we have to show for a reachable state $s = (p,m)$, i.e. $s \in ReachableFrom\ (effS_B\ \Pi)\ (Starters\ \Pi)$, that it is safe and satisfies the annotation $aF\ \Pi\ p = ispecF\ \Pi\ p\ d$. The proof is by induction on *ReachablesFrom*. The base case $s \in Starters\ \Pi$ is trivial due to the definition of *Starters*. In the inductive case we have a predecessor state $s' = (p',m')$, such that $(s',s) \in (effS_B\ \Pi)$ and $\Pi,s' \models ispecF\ \Pi\ p\ d$. Note that p' can only be initial or internal, but not final. In case p' is internal we apply an auxiliary lemma: $\ldots \wedge diameter\ (effS_B\ \Pi)\ d \wedge (p',m') \models ispecF\ \Pi\ p'\ d \longrightarrow \Pi,(p',m') \models ispecF\ \Pi\ p'\ (d+1)$, which we prove by induction on *ReachablesFromIn*. The underlying idea is that when we exceed the diameter we reach a state that is also reachable with fewer steps and can thus apply our induction hypothesis. We also use another auxiliary lemma, which ensures that *ispecF* is preserved. Provided we have $\Pi,s' \models ispecF\ \Pi\ p\ k$ then all follow up states s'' reachable from s' in l steps satisfy $\Pi,s'' \models ispecF\ \Pi\ p''\ (k - l)$. We prove it by induction on the difference $k - l$ and need requirement 2.18. From the first lemma we obtain $\Pi,s' \models ispecF\ \Pi\ p'\ (d+1)$ and finish the proof by unfolding *ispecF* and applying requirement 2.19. In case p' is initial, we have $\Pi,s' \models ispecF\ \Pi\ p'\ d'$ for any d' any can apply the second lemma to finish the proof. □

2.9 Invariant Verification Conditions

Theorem 2.3 only states something for strongly annotated programs. What can we say about the verification conditions if annotations are only correct?

locale *invariantVCG* = *VCG* +

Theorem 2.5 *(invariantVCG)* For wellformed, safe and correctly annotated programs the verification condition holds for all reachable states.

wf $\Pi \wedge isSafe\ \Pi \wedge correctAn\ \Pi \longrightarrow (\forall\, s \in Reachables\ \Pi.\ \Pi,s \models vcg\ \Pi)$

Showing that the verification condition is an invariant is enough to guarantee safety for correctly annotated programs. This follows from Theorem 2.1 with Requirement 2.10 and Lemma 2.2. In our example Π_x the annotation $A_0 = (x = 0)$ is correct and we can expect an invariant as verification condition. However, when we try to show $\forall s \in$ *Reachables* Π_x. $\Pi_x,s \models vcg\ \Pi_x$, we might have to derive $y = 0$ from the operational semantics. The set *Reachables* Π is a semantical notion of the strongest invariant and contains all valid facts for reachable states. This shows the drawback we have with annotations that are "only" correct. For strongly annotated programs the verification condition is a tautology. In tautologous verification conditions all information needed to show validity is already inside and no facts needs to be derived from the operational semantics.

Nevertheless Theorem 2.5 is interesting as it says something about the VCG in case of weaker annotations. To prove it, we need requirements that are quite similar to the completeness requirements. Additional premises, such as $(p,m) \in$ *Reachables* Π make these requirements easier to instantiate than their counterparts in locale *completeVCG*.

Requirement 2.20 *wf* $\Pi \wedge$ *correctAn* $\Pi \wedge$ *isSafe* $\Pi \wedge (p,m) \in$ *Reachables* $\Pi \wedge$
$\Pi,(p,m) \models B \wedge (p',B) \in$ *set* (*succsF* $\Pi\ p'') \longrightarrow\ (p=p'' \wedge (\exists\ m'.\ ((p,m),(p',m')) \in$ (*effS* $\Pi)))$

Requirement 2.21 *wf* $\Pi \wedge$ *correctAn* $\Pi \wedge$ *isSafe* $\Pi \wedge\ (p,m) \in$ *Reachables* $\Pi \wedge (\exists\ B.\ (p',B)$
$\in$ *set* (*succsF* $\Pi\ p)) \wedge$
$((p,m),(p',m')) \in$ (*effS* $\Pi) \wedge \Pi,(p',m') \models Q \longrightarrow \Pi,(p,m) \models$ *wpF* $\Pi\ p\ p'\ Q$

In addition to these *invariantVCG* also has Requirements 2.15, 2.12 and 2.13 just as *completeVCG* does. The proof for Theorem 2.5 is similar to the one for Theorem 2.3. Only the premises need to be adjusted, the induction and the auxiliary lemmas stay the same.

2.10 Instantiating the Framework

Our framework distributes over multiple theories and locales as shown in Fig. 2.2. Definitions and theorems inside locales are relative to the assumptions made. To make use of locales one has to instantiate them, which involves providing definitions for all parameters and proving that they meet the requirements. For each locale Isabelle/HOL automatically generates a predicate taking all parameters as arguments. For example for the locale *correctVCG* we get the following predicate:

correctVCG initS effS $\lfloor T \rfloor$ $\lfloor F \rfloor$ $\lfloor \bigwedge \rfloor$ $\lfloor \Rightarrow \rfloor \models \vdash$ *ipc anF succsF wf initF wpF*

This predicate places the given parameters into a big conjunction of all the requirements.

Note that the predicate *correctVCG*, just as all the other locales, does not depend on *safeF*. This means the safety policy *safeF* does not matter for the instantiation and can be replaced without having to adjust proofs. To alter other parameters, our framework provides instantiation theorems. For example, the following theorem allows to replace the wellformedness checker *wf* with a stronger version *wf'*.

Theorem 2.6 Severing wellformedness conditions preserves a VCG's correctness.

$$\frac{\begin{array}{c}\forall\,\Pi.\ wf'\ \Pi \longrightarrow wf\ \Pi \\ correctVCG\ initS\ effS\ \llbracket T \rrbracket\ \llbracket F \rrbracket\ \bigwedge\!\!\!\bigwedge\ \Rightarrow\!\!\!\!\Rightarrow\ \models \vdash\ ipc\ anF\ succsF\ wf\ initF\ wpF\end{array}}{correctVCG\ initS\ effS\ \llbracket T \rrbracket\ \llbracket F \rrbracket\ \bigwedge\!\!\!\bigwedge\ \Rightarrow\!\!\!\!\Rightarrow\ \models \vdash\ ipc\ anF\ succsF\ wf'\ initF\ wpF}$$

The benefit of such instantiation theorems is that one can upgrade an instantiation later on without having to prove all requirements again. This is in particular important for the successor function *succsF*, which we will upgrade later on to integrate trusted facts from external program analysers. For this purpose we use the functional *upg*. It modifies a given successor function by conjoining formulas from a given assignment *iF* to its branch conditions:

upg:: $('prog \Rightarrow 'pos \Rightarrow 'form) \Rightarrow\ ('prog \Rightarrow 'pos \Rightarrow ('pos \times 'form)\ list) \Rightarrow ('prog \Rightarrow 'pos \Rightarrow ('pos \times 'form)\ list)$

$upg\ iF\ sucF = \lambda\Pi\ p.\ map\ (\lambda(p',\ B).\ (p',\ \bigwedge\!\!\!\bigwedge\ [B,\ iF\ \Pi\ p]))\ (sucF\ \Pi\ p)$

Equipped with *upg*, we can improve the quality of branch conditions of a given successor function *succsF*. Here is an example, in which we augment *succsF* with the formulas *iF* yields for program positions.

$succsF\ \Pi\ p = [(p_1,B_1),\ldots,(p_k,B_k)] \longrightarrow$
$(upg\ iF\ succsF)\ p = [(p_1,B_1 \mathbin{\triangle\!\!\!\triangle} iF\ p),\ \ldots,\ (p_k,B_k \mathbin{\triangle\!\!\!\triangle} iF\ p)]$

The following theorem states that correctness of the VCG stays intact if one upgrades the successor function with invariants.

Theorem 2.7 Upgrading *succsF* with an invariant *iF* preserves a VCG's correctness.

$$\frac{\begin{array}{c}\forall\,\Pi.\ wf\ \Pi \longrightarrow (\forall\, s{\in}ReachableFrom\ (effS\ \Pi)\ (initS\ \Pi).\ \Pi,s \models iF\ \Pi\ (fst\ s)) \\ succsF' = upg\ iF\ succsF \\ correctVCG\ initS\ effS\ \llbracket T \rrbracket\ \llbracket F \rrbracket\ \bigwedge\!\!\!\bigwedge\ \Rightarrow\!\!\!\!\Rightarrow\ \models \vdash\ ipc\ anF\ succsF\ wf\ initF\ wpF\end{array}}{correctVCG\ initS\ effS\ \llbracket T \rrbracket\ \llbracket F \rrbracket\ \bigwedge\!\!\!\bigwedge\ \Rightarrow\!\!\!\!\Rightarrow\ \models \vdash\ ipc\ anF\ succsF'\ wf\ initF\ wpF}$$

Note that the Theorems 2.6 and 2.7 are proven outside any locale and thus do not depend on further assumptions.

2.11 Conclusion

Our framework makes various contributions for constructing trustworthy PCC systems. First, it identifies the program semantics, safety logic and safety policy as the three major influences on PCC, and gives a blueprint for a PCC architecture where these factors are kept modular. It defines a VCG that can easily be adapted to different PCC setups by adjusting its parameter functions.
Second, because the VCG is written in an executable style one can use Isabelle's code generator to obtain an ML program. Other features of Isabelle/HOL like its support for proof objects and checking, or tactics and decision procedures for proof generation allow to simulate the full work flow of a PCC system inside the theorem prover.

Third, it makes requirements on the parameters explicit and proves the VCG correct and relatively complete. The completeness theorems only regard the semantical properties of verification conditions. Depending on the quality of annotations, they are either tautologies or just invariants. In contrast to Hoare Logic, our verification conditions also give guarantees for non-terminating programs and our VCG works for programs with arbitrarily "structured" control flow. The fact that our framework's theorems carry over to all instantiations makes it an ideal starting point for prototyping PCC systems. One can alter the parameters, rerun the proofs and discover how changes affect the soundness or performance of the modified system.

In this thesis we will instantiate this framework to Jinja [55] bytecode. We also have instantiations for simple assembly languages [98, 96]. Although these have by no means the complexity of Jinja, they are mature enough to verify interesting examples [4] with non-trivial safety policies. Apart from policies that prevent arithmetic overflow, we also verified examples with type safety or bounds on memory usage as safety policies. For example in [98] we verified that a little smart card purse does not overflow. Apart from that we also verified that in-place list reversal does not consume additional memory and terminates within a certain number of instruction executions. To demonstrate that we can deal with recursion, we also verified that a recursive multiplication algorithm is functionally correct and does not overflow.

3 Jinja Bytecode and Virtual Machine

This chapter defines the syntax of Jinja bytecode programs and their semantics with a formalised virtual machine. We also introduce an example program that will serve for illustration throughout the thesis.

3.1 Jinja Bytecode

Jinja bytecode is a down-sized version of Java bytecode. Despite its small instruction set, it covers most object oriented features: objects, dynamic method calls and exceptions.

datatype *instr* =

`Load` *nat*	load from register
| `Store` *nat*	store into register
| `Push` *val*	push a constant
| `New` *cname*	create object on heap
| `Getfield` *vname cname*	fetch field from object
| `Putfield` *vname cname*	set field in object
| `Checkcast` *cname*	check if object is of class *cname*
| `Invoke` *mname nat*	invoke method with *nat* parameters
| `Return`	return from method
| `Pop`	remove top element
| `IBin` *num-op*	integer arithmetic
| `Goto` *int*	goto relative address
| `CmpEq`	equality comparison
| `IfFalse` *int*	branch if top of stack false
| `IfIntCmp` *rel-op int*	take integers a and b from stack, branch on *relop rel a b*.
| `Throw`	throw exception

Figure 3.1: Jinja bytecode instructions

For the sake of simplicity, Jinja does not have static methods and fields, non default

constructors, arrays and threads. In total the Java VM supports about 200 instructions. Many of these just perform similar tasks. We tried to reduce this high number, which leads to many case splits in proofs, by having multi-purpose instructions. In particular, integer arithmetic and branches are subsumed by two instructions only. Mode arguments of type *num-op* or *rel-op* distinguish various operations.

num-op = *Add* | *Sub* | *Mul*
rel-op = *Less* | *Leq* | *Eq* | *Geq* | *Grtr*
cname = *vname* = *mname* = *string*

With *numop* and *relop* we connect these symbols to Isabelle/HOL's arithmetic.

numop :: *num-op* ⇒ *int* ⇒ *int* ⇒ *int*

numop Add a b = (*a* + *b*)
numop Sub a b = (*a* − *b*)
numop Mul a b = (*a* ∗ *b*)

relop :: *rel-op* ⇒ *int* ⇒ *int* ⇒ *bool*

relop Less a b = (*a* < *b*)
relop Leq a b = (*a* ≤ *b*)
relop Eq a b = (*a* = *b*)
relop Geq a b = (*a* ≥ *b*)
relop Grtr a b = (*a* > *b*)

Both functions are used in the definition of the operational semantics (§3.2) and for the semantics of assertions (§4). For better readability we abbreviate some instructions as follows:

`IAdd` = `IBin` *Add* `IfIntL` = `IfIntCmp` *Less* `IfIntGeq` = `IfIntCmp` *Geq*
`ISub` = `IBin` *Sub* `IfIntLeq` = `IfIntCmp` *Leq* `IfIntG` = `IfIntCmp` *Grtr*
`IMul` = `IBin` *Mul* `IfIntEq` = `IfIntCmp` *Eq*

The original Jinja bytecode [55] only has one arithmetic instruction and one conditional branch. In order to translate real Java programs to Jinja, we adjusted all the definitions and proofs in [55] to this extended instruction set.

Jinja has values for booleans, e.g. *Bool True*, integers, e.g. *Intg 5*, references, e.g. *Addr 3*, null pointer, e.g. *Null* or a dummy element, e.g. *Unit*.

datatype *val* = *Bool bool* | *Intg int* | *Addr addr* | *Null* | *Unit*

For booleans and integers we use the corresponding Isabelle/HOL types. Addresses are modelled as natural numbers.

types *addr* = *nat*

For some values, we introduce destructor functions. In case these are applied to improper values, their result becomes *arbitrary*, a default value Isabelle/HOL provides automatically for each type, i.e. *the-Intg* (*Bool b*) = *arbitrary*.

the-Intg (*Intg i*) = *i*, *the-Bool* (*Bool b*) = *b*, *the-Addr* (*Addr a*) = *a*

Each value has a type associated with it:

datatype *ty* = *Boolean* | *Integer* | *Class cname* | *NT* | *Void*

With *liftI* and *liftR* we lift Isabelle/HOL's arithmetic or relational operators, e.g. +, −, <, = and so on, to Jinja values. In case any argument has improper type the result becomes *None*.

liftI :: ((*int* ⇒ *int* ⇒ *int*) × *val option* × *val option*) ⇒ *val option*
liftI (*f*,⌊*Intg a*⌋,⌊*Intg b*⌋) = ⌊*Intg* (*f a b*)⌋
liftI (*f*,*oth*,*oth*′) = *None*

liftR :: ((*int* ⇒ *int* ⇒ *bool*) × *val option* × *val option*) ⇒ *val option*
liftR (*r*,⌊*Intg a*⌋,⌊*Intg b*⌋) = ⌊*Bool* (*r a b*)⌋
liftR (*r*,*oth*,*oth*′) = *None*

To illustrate how Jinja bytecode looks like Fig. 3.3 shows a translation of the Java program shown in Fig. 3.2. The example has three classes `Start`, `Cnt` and `No`. The first contains the *main* method, which creates a new counter object, sets it to `x0` and then adds a constant `y0` to it by calling `up`. The `up` method of class `Cnt` adds its argument to counter field `c` or throws a `No` exception in case of a negative or too large argument. Depending on how `up` terminates `x` either becomes `x0 + y0`, `0` or `-1`. The specifications in Fig. 3.2 are written in JML [58]. We will use similar annotations in our assertion logic (see §4) to verify that the program does not cause arithmetic overflows, not matter how we chose the initial constants `x0` and `y0`. If a programmer slightly modifies the check in method `up`, for example by swapping the disjunction to `(maxI - i < c || i < z)`, then the check itself can cause an overflow. The check in Fig. 3.3 is correct, because the left hand side of the disjunction `||` is checked first in the bytecode. This means `i` is non-negative in the check of the right hand side. We subtract two non-negative numbers from each other, which never produces an over- or underflow. Also if we change the check to `(i < z || maxI < c + i)`, the check itself can cause an overflow and also the addition it should protect. This illustrates that some bugs can be found at the bytecode level, where expression evaluation is ordered, but not so easy at the source level.

Jinja bytecode identifies instructions with positions. These are triples of type *cname* × *mname* × *nat*. For example (*C*,*M*,*pc*) points to instruction number *pc* in method *M* of class *C*. Each method is a tuple of the form (*mxs*,*mxr*,*is*,*et*), where *mxs* indicates the maximum operand stack height, *mxr* the number of usable registers, *is* the instructions of the method body and *et* the exception table.

types *jvm-method* = *nat* × *nat* × *instr list* × *ex-table*

```
class Cnt {

  static final int maxI =
  Integer.MAX_VALUE;

  int c;


  //@ ensures c = 0;
  void reset() {
  c = 0; }


  /*@ ensures c = s;
    @ assignable c;
    @ signals (No ne)
    @ c = 0 && s < 0 ;
    @*/
  void set(int s)
  throws No {
  reset();
  up(s); }


  /*@ ensures
    @ c = \old(c) + \old(i)
    @ && \result = c;
    @ assignable c;
    @ signals (No ne)
    @ c = \old(c) &&
    @ (i < 0 || maxI - i < c)
    @*/
  int up(int i)
  throws No {
  int z = 0;
  if (i < z || maxI - i < c)
  then throw No();
  c = c + i;
  return c; }
}
```

```
class Start {

 static final int x0 = 3;
 static final int y0 = 15;

 void main() {
 int x = x0;
 int y = y0;
 try { Cnt ct = new Cnt();
       /*@ assert
         @ x = x0 && y = y0 @*/
       ct.set(x);
       /*@ assert x = x0 &&
         @ y = y0 && ct.c = x0 @*/
       x=ct.up(y);
       /*@ assert x = ct.c &&
         @ y = y0 && ct.c = x0 + y0 @*/
      }
 catch (No ne) {


  /*@ assert
    @ x = x0 && y = y0 @*/
  x=0; }
 catch (Exception e) {


  /*@ assert
    @ x = x0 && y = y0 @*/
  x=-1; }
 }
}

class No {
}
```

Figure 3.2: A Java Counter

```
class Cnt {

Void reset () {
0 Load 0
1 Push Intg 0
2 Putfield c Cnt
3 Push Unit
4 Return }

Void set (Integer) {
0 Load 0
1 Invoke reset 0
2 Pop
3 Load 0
4 Load 1
5 Invoke up  1
6 Pop
7 Push Unit
8 Return }

Integer up (Integer) {
0 Push Intg 0
1 Store 2
2 Load 1
3 Load 2
4 IfIntL 7
5 Push (Intg 2147483647)
6 Load 1
7 ISub
8 Load 0
9 Getfield c Cnt
10 IfIntGeq 5
11 New No
12 Push Null
13 Pop
14 Throw
15 Load 0
16 Load 0
17 Getfield c Cnt
18 Load 1
19 IAdd
20 Putfield c Cnt
21 Load 0
22 Getfield c Cnt
23 Return }
}
```

```
class Start {

Void main () {
0 Push (Intg 3)
1 Store 1
2 Push (Intg 15)
3 Store 2
4 New Cnt
5 Push Null
6 Pop
7 Store 3
8 Load 3
9 Load 1
10 Invoke set 1
11 Pop
12 Load 3
13 Load 2
14 Invoke up 1
15 Store 1
16 Push Unit
17 Return
18 Store 3
19 Push (Intg 0)
20 Store 1
21 Push Unit
22 Return
23 Store 3
24 Push (Intg -1)
25 Store 1
26 Push Unit
27 Return

from 4 to 16 catch No at 18
from 4 to 16 catch Exception at 23 }

}
```

Figure 3.3: Counter in Jinja Bytecode

The exception table is a list of tuples $(f,\ t,\ E,\ h,\ d)$:

types *ex-table* = $(nat \times nat \times cname \times nat \times nat)\ list$

Whenever an instruction within the *try* block ranging from f to t, i.e. $[f,\ t)$, throws an exception of type *Class* E the handler starting at h is executed. If an exception occurs, for which the current method has no matching handler, control is transferred to the caller method, where the search for a handler continues. The parameter d, which is always *0* in our case, specifies how many additional values the handler expects on the operand stack upon entry. This is used in [55] to handle exceptions within expression evaluation, but is not required for bytecode obtained from Java programs. In Java, exceptions can only be caught at the statement level.

Jinja programs are lists of class declarations. Each class declaration (C,S,fs,ms) consists of the name of the class C, the name of its direct superclass S, a list of field declarations fs, which are pairs of field names and types, and a list of method declarations ms. Method declarations $(M,aTys,rTy,bd)$ consist of the method's name M, its argument types $aTys$, its result type rTy and its body bd.

types *jvm-prog* = $(cname \times cname \times fdecl\ list \times mdecl\ list)\ list$
$fdecl = vname \times ty$
$mdecl = mname \times ty\ list \times ty \times$ *jvm-method*

We write *method P C M* (see [55]) to lookup the method with name M that is visible from C in a program P. The result is of the form (D,Ts,T,m), where D is the hierarchically closest class from C that declares a method with name M and (M,Ts,T,m) is the declaration of this method. Analogously we write *field P C F* (see [55]) to fetch field declarations. It gives us (D,T), where D is the closest class that declares a field with name F and T is the type of that field.

Our PCC system requires programs Π with annotations, which we give in form of a finite map from positions to logical expressions.

types *jbc-prog* = *jvm-prog* $\times$ $(pos \mathrel{\sim\!\!\!>} expr)$

3.2 Operational Semantics

Jinja programs are interpreted by the Jinja virtual machine, which closely models the Java VM.

3.2.1 States

We model Jinja states σ as triples (x,h,frs). They consist of a flag x indicating whether an exception is raised, a heap h, and a method frame stack frs.

types *jvm-state* = *addr option* × *heap* × *frame list*

In case an exception occurs the flag records a reference to the corresponding exception object. The heap is a partial map from addresses (natural numbers) to objects.

types *heap* = *addr* ⇒ *obj option*
obj = *cname* × *fields* *fields* = (*vname* × *cname*) ⇒ *val option*

Whenever a method is invoked, a new frame is allocated on the method frame stack. This frame contains an operand stack, registers and the program counter. In the registers the Jinja VM stores the `this` reference, the method's arguments and its local variables. The operand stack is used to evaluate expressions.

types *frame* = *opstack* × *registers* × *pos*
opstack = *val list* *registers* = *val list* *pos* = *cname* × *mname* × *nat*

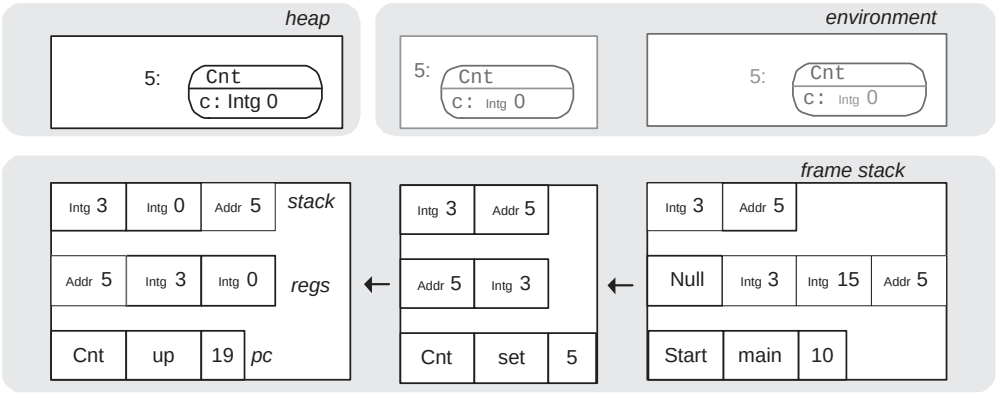


Figure 3.4: Jinja VM Snapshot

3.2.2 Extended Machine

Some constructs of our assertion language (§4) require extended Jinja states (p,σ,e), which carry additional information in an environment e. Since our framework expects states as tuples $'pos \times 'mem$, the first component of a *jbc-state* is a position.

$$\frac{\begin{array}{l} P = fst\ \Pi \quad p = (C,\ M,\ pc) \\ i = (instrs\text{-}of\ P\ C\ M)_{[pc]} \\ \sigma = (None,\ h,\ (st,\ rg,\ p)\cdot frs) \\ exec\text{-}instr\ i\ P\ h\ st\ rg\ C\ M\ pc\ frs \\ \quad = (None,h',fr'\cdot frs') \\ \sigma' = (None,\ h',\ fr'\cdot frs') \\ p' = snd\ (snd\ fr') \\ e' = e(\!|cs{:=}\textit{if}\ \exists M\ n.\ i = \\ \texttt{Invoke}\ M\ n\ \textit{then}\ h\cdot(cs\ e) \\ \textit{else if}\ i = \texttt{Return} \\ \quad \textit{then}\ tl\ (cs\ e)\ \textit{else}\ cs\ e|\!) \end{array}}{((p,\ \sigma,\ e),\ p',\ \sigma',\ e') \in effS\ \Pi}\ \text{NRML}$$

$$\frac{\begin{array}{l} P = fst\ \Pi \quad p = (C,\ M,\ pc) \\ i = (instrs\text{-}of\ P\ C\ M)_{[pc]} \\ \sigma = (None,\ h,\ (st,\ rg,\ p)\cdot frs) \\ exec\text{-}instr\ i\ P\ h\ st\ rg\ C\ M\ pc\ frs = \\ \quad (\lfloor xa\rfloor,h',\_) \\ find\text{-}handler\ P\ xa\ h\ ((st,rg,p)\cdot frs) \\ \quad = \sigma' \\ \sigma' = (None,h, \\ \quad ([Addr\ xa],rg',p')\cdot frs') \\ e' = \ e(\!|cs{:=}drop\ (|frs| \\ \quad - |frs'|)\ (cs\ e)|\!) \end{array}}{((p,\ \sigma,\ e),\ p',\ \sigma',\ e') \in effS\ \Pi}\ \text{EXPT}$$

Figure 3.5: Semantics of the extended Jinja VM

types *jbc-state* = *pos* × *jvm-state* × *env*

The position is also stored in σ, and we write *pos* σ to access it.

pos :: *jvm-state* ⇒ *pos*
pos (*x*,*h*,(*st*,*rg*,*p*)·*frs*) = *p*

This redundancy is unpleasant for data modeling, but harmless for our verifications. For reachable states (p,σ,e), we always have p = *pos* σ.
The environment e contains a virtual stack of call states *cs* e and a binding *lv* e for so-called logical variables, which we introduce in §4.

record *env* = *cs*:: *heap list* *lv*:: *var* ⇒ *val*

Whenever a new frame is allocated on the method frame stack, we record the current heap on the call stack, which acts like a history variable in Hoare Logics. Whenever a frame is popped, we also pop an entry from the call stack. In Fig. 3.4 we show a snapshot of the extended virtual machine. The machine executes instruction `IAdd` next, which removes *Intg 3* and *Intg 0* from the operand stack and pushes *Intg 3* back. Formally, we define the semantics of the extended machine with the two rules shown in Fig. 3.5. One rule specifies normal, the other one exceptional execution. For the state transformation inside the *jvm-state* component of *jbc-state* we use functions from the original Jinja VM [55].
In both rules we use *instrs-of* (see [55]) to retrieve the instruction list of the current method. The actual execution for single instructions is delegated to *exec-instr*.

exec-instr :: [*instr*, *jvm-prog*, *heap*, *val list*, *val list*, *cname*, *mname*, *pc*, *frame list*] => *jvm-state*

$$\begin{array}{l}
\textit{exec-instr}\ (\texttt{Load}\ n)\ P\ h\ st\ rg\ C_0\ M_0\ pc\ frs = \\
(\textit{None},\ h,\ (rg_{[n]}\ \cdot\ st,\ rg,\ C_0,\ M_0,\ pc{+}1)\cdot frs) \\
\\
\textit{exec-instr}\ (\texttt{Store}\ n)\ P\ h\ st\ rg\ C_0\ M_0\ pc\ frs = \\
(\textit{None},\ h,\ (tl\ st,\ rg[n{:=}hd\ st],\ C_0,\ M_0,\ pc{+}1)\cdot frs) \\
\\
\textit{exec-instr}\ (\texttt{Push}\ v)\ P\ h\ st\ rg\ C_0\ M_0\ pc\ frs = \\
(\textit{None},\ h,\ (v\ \cdot\ st,\ rg,\ C_0,\ M_0,\ pc{+}1)\cdot frs) \\
\\
\textit{exec-instr}\ \texttt{Pop}\ P\ h\ st\ rg\ C_0\ M_0\ pc\ frs = \\
(\textit{None},\ h,\ (tl\ st,\ rg,\ C_0,\ M_0,\ pc{+}1)\cdot frs)
\end{array}$$

Figure 3.6: Argument Passing

The parameters of *exec-instr i P h st rg* C_0 M_0 *pc frs* are the following: the instruction to execute, the program P, the heap h, the operand stack *st* and registers *rg* of the current frame, the class C_0 and name M_0 of the method that is currently executed, the current *pc*, and the rest of the call frame stack *frs*.
The results of *exec-instr* are triples, whose first component indicates whether an exception occurs. If yes (rule EXPT), we use the function *find-handler* (see [55]) to do exception handling similar to the Java VM: it looks up the exception table in the current method, and sets the program counter to the first handler that protects *pc* and that matches the exception class. If there is no such handler, the topmost call frame is popped, and the search continues recursively in the invoking frame. If no exception handler is found, the machine halts. If this procedure does find an exception handler $(f,t,C,h,0)$ it sets the *pc* to h and empties the operand stack except for the reference to the exception object *xa*. Next, we reveal the details of *exec-instr*, whose definition can also be found in the Jinja article [55], except for the new arithmetic and branch instructions.

3.2.3 Argument Passing

In Fig. 3.6 we have the semantics of instructions that pass arguments between registers and the operand stack. With `Load` n the machine pushes register n onto the stack. The instruction `Store` n does exactly the opposite. To push a constant value v onto the stack, we have `Push` v. With `Pop` we remove the topmost stack value.

3.2.4 Arithmetics, Checks and Branches

Instructions performing binary arithmetic operations, checks or branches are defined in Fig. 3.7. The instruction `IBin` *no* expects two integers on top of the stack. It removes both, applies operation *numop op* (see §3.1) on them and pushes the result

back. With `IfIntCmp` *ro t* the machine checks if the two topmost integers on the stack satisfy the relation *relop ro* (see §3.1) and removes these. If so it jumps *t* instructions forward (backward if *t* is negative), otherwise just one. The instruction `Goto` *t* jumps *t* instructions forward. The instruction `CmpEq` just checks whether the two topmost stack values are equal. It removes both and pushes back a boolean with the result. This result *b* can then be checked and removed with `IfFalse` *t*, which jumps *t* instructions forward if *b* is *Bool False*. To check the dynamic type of objects the machine provides `Checkcast` *Cl*. It expects an object reference *r* on top of the stack and checks whether it has a subtype of *Cl*. In this case, the function *cast-ok P C h v* (see [55]) yields true and `Checkcast` silently removes the reference. If not, it throws a *ClassCast* exception. This is one of the pre-allocated system exceptions, whose address is determined by the function *addr-of-sys-xcpt* (see [55]). With `Throw`, one can raise exceptions explicitly. It expects an object reference on top of the stack and simply sets the exception flag. If the reference is *Null*, it fails with a *NullPointer* exception.

3.2.5 Heap Access

In Fig. 3.8 we show the instructions that read from or modify the heap. The instruction `Getfield` *F C* expects a reference to an object of type *C* on top of the stack. It replaces this reference with the value of this object's field *F*. In case the reference is *Null*, it fails with a *NullPointer* exception. The `Putfield` *F C* instruction expects a value and a reference to an object of type *C* on top of the stack. It removes both and updates the referenced object field with this value. In case the reference is *Null*, it raises a *NullPointer* exception. With *New C* the machine creates a new object of type *C*, initialises its fields with dummy values and pushes a reference to this object onto the stack. The creation and initialisation of the new object is handled by *blank P C* (see [55]), its address is determined by *new-Addr h* (see [55]). In case the heap is full, *new-Addr h* yields *None* and `New` *C* raises an *OutOfMemory* exception.

3.2.6 Method Invocation and Return

Finally, Fig. 3.9 shows how methods are invoked and how they return. The instruction `Invoke` *M n* expects *n+1* arguments on top of the stack. The bottom one must be a reference to an object whose class provides or inherits a method with name *M* taking *n* arguments. Here, we abstract from the real JVM [88], where methods are actually identified by their name and argument types. In case the reference is *Null*, the machine throws a *NullPointer* exception. Otherwise, it fetches the invoked method's body with *method P C M* and allocates a new frame on the frame stack. This frame has an empty

```
exec-instr (IBin no) P h st rg C0 M0 pc frs =
 (let i2 = the-Intg (hd st);
      i1 = the-Intg (hd (tl st))
  in (None, h, (Intg (numop no (i1) (i2)) · (tl (tl st)), rg, C0, M0, pc+1) · frs))

exec-instr (IfIntCmp ro t) P h st rg C0 M0 pc frs =
(let i2 = the-Intg (hd st);
     i1 = the-Intg (hd (tl st))
in (None, h, ((tl (tl st)), rg, C0, M0, (if relop ro i1 i2
              then nat(int pc+t) else pc+1)) · frs))

exec-instr (Goto i) P h st rg C0 M0 pc frs =
(None, h, (st, rg, C0, M0, nat(int pc+i)) · frs)

exec-instr CmpEq P h st rg C0 M0 pc frs =
(let v2 = hd st;
     v1 = hd (tl st)
 in (None, h, (Bool (v1=v2) · tl (tl st), rg, C0, M0, pc+1) · frs))

exec-instr (IfFalse i) P h st rg C0 M0 pc frs =
(let pc' = if hd st = Bool False then nat(int pc+i) else pc+1
 in (None, h, (tl st, rg, C0, M0, pc') · frs))

exec-instr (Checkcast C) P h st rg C0 M0 pc frs =
(let r   = hd st;
      xp' = if ¬cast-ok P C h r then ⌊addr-of-sys-xcpt ClassCast⌋ else None
in (xp', h, (st, rg, C0, M0, pc+1) · frs))

exec-instr Throw P h st rg C0 M0 pc frs =
(let xp' = if hd st = Null then ⌊addr-of-sys-xcpt NullPointer⌋
           else ⌊the-Addr(hd st)⌋
 in (xp', h, (st, rg, C0, M0, pc) · frs))
```

Figure 3.7: Arithmetics, Checks and Branches

operand stack. The initial registers contain the `this` reference at location *0* and the method's arguments at the locations *1* to *n*. The remaining registers are used to store local variables and are initialised with *arbitrary*. Here we use the function *replicate k v*, which yields a list of *v* elements of length *k*. In Fig. 3.10 we illustrate how the program from Fig. 3.3 invokes the method *up* from method *main*.
A method returns to its call frame when a `Return` instruction occurs. The topmost value of the operand stack is the method's result and is pushed onto the call frame's operand stack, after all the arguments have been cleared. In Fig. 3.11 we show how method *up* returns to *main*.

```
exec-instr (Getfield F C) P h st rg C0 M0 pc frs =
(let v      = hd st;
     xp'    = if v=Null then ⌊addr-of-sys-xcpt NullPointer⌋ else None;
     (D,fs) = the(h(the-Addr v))
in (xp', h, (the(fs(F,C))·(tl st), rg, C0, M0, pc+1)·frs))

exec-instr (Putfield F C) P h st rg C0 M0 pc frs =
(let v    = hd st;
     r    = hd (tl st);
     xp'  = if r=Null then ⌊addr-of-sys-xcpt NullPointer⌋ else None;
     a    = the-Addr r;
     (D,fs) = the (h a);
     h'   = h(a ↦ (D, fs((F,C) ↦ v)))
in (xp', h', (tl (tl st)), rg, C0, M0, pc+1)·frs))

exec-instr (New C) P h st rg C0 M0 pc frs = (case new-Addr h
of None ⇒ (⌊addr-of-sys-xcpt OutOfMemory⌋, h, (st, rg, C0, M0, pc)·frs)
| ⌊a⌋ ⇒ (None, h(a↦blank P C), (Addr a·st, rg, C0, M0, pc+1)·frs))
```

Figure 3.8: Heap Access

```
exec-instr (Invoke M n) P h st rg C0 M0 pc frs =
(let ps  = take n st;
     r   = st[n];
     xp' = if r=Null then ⌊addr-of-sys-xcpt NullPointer⌋ else None;
     C   = fst(the(h(the-Addr r)));
     (D,M',Ts,mxs,mxl0,ins,xt)= method P C M;
     f'  = ([],[r]@(rev ps)@(replicate mxl0 arbitrary),D,M,0)
in (xp', h, f'·(st, rg, C0, M0, pc)·frs))

exec-instr Return P h st0 rg0 C0 M0 pc frs =
(if frs=[] then (None, h, []) else
   let v = hd st0;
   (st,rg,C,m,pc) = hd frs;
   n = length (fst (snd (method P C0 M0)))
in (None, h, (v·(drop (n+1) st),rg,C,m,pc+1)·tl frs))
```

Figure 3.9: Method Invocation and Return

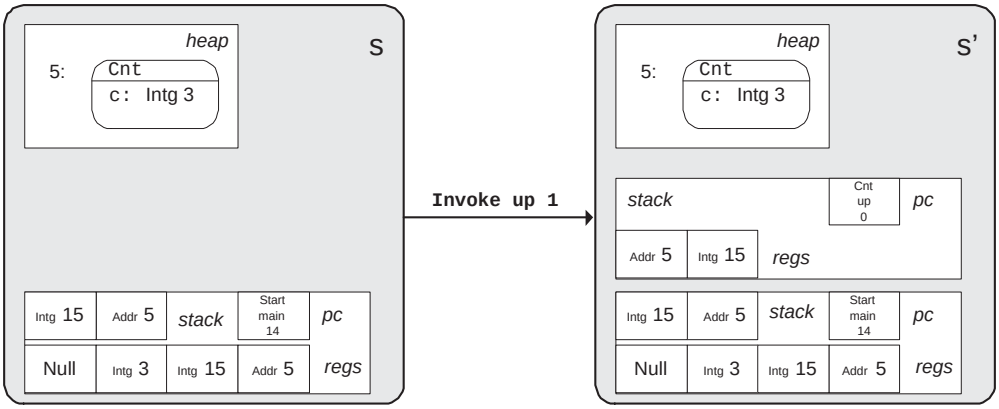


Figure 3.10: Invoke up 1

3.2.7 Initial States

We require Jinja Bytecode programs to have a *main* method. For simplicity we assume that this method is named *main*, has no arguments, and belongs to a class called *Start*. This means we start a program at position (*Start*,*main*,*0*). The initial operand stack is empty and the registers are initialised with arbitrary values. The initial heap (*start-heap* (*fst* Π)) contains three entries for system exceptions *NullPointer*, *ClassCast* and *OutOfMemory*. The initial environment contains an empty call stack and an unrestricted binding of logical variables.

$initS\ \Pi = \{(p,\ \sigma,\ e).\ p = (Start,\ main,\ 0) \wedge cs\ e = [] \wedge (\mathsf{let}\ (\_,\ \_,\ (\_,\ \_,\ (mxr,\ \_))) = method\ (fst\ \Pi)\ Start\ main\ \mathsf{in}\ \sigma = (None,\ start\text{-}heap\ (fst\ \Pi),\ [([],\ Null \cdot replicate\ mxr\ arbitrary,\ p)]))\}$

3.2.8 Simulation

The operational semantics *effS* describes the behaviour of our extended Jinja VM. In addition to "normal" states of type *jvm-state* it also maintains an environment *env*. The original Jinja VM [55] is defined as a relation $P \vdash \sigma \xrightarrow{\text{jvm}}_1 \sigma'$ on normal states only:

$- \vdash - \xrightarrow{\text{jvm}}_1 - :: jvm\text{-}prog \Rightarrow (jvm\text{-}state \times jvm\text{-}state)\ set$

Although the definition of $P \vdash \sigma \xrightarrow{\text{jvm}}_1 \sigma'$ uses the same auxiliary functions, namely *exec-instr*, *find-handler* and so on, it behaves slightly differently. In Jinja [55] exception handlers can be entered with additional arguments on the operand stack. If so, the num-

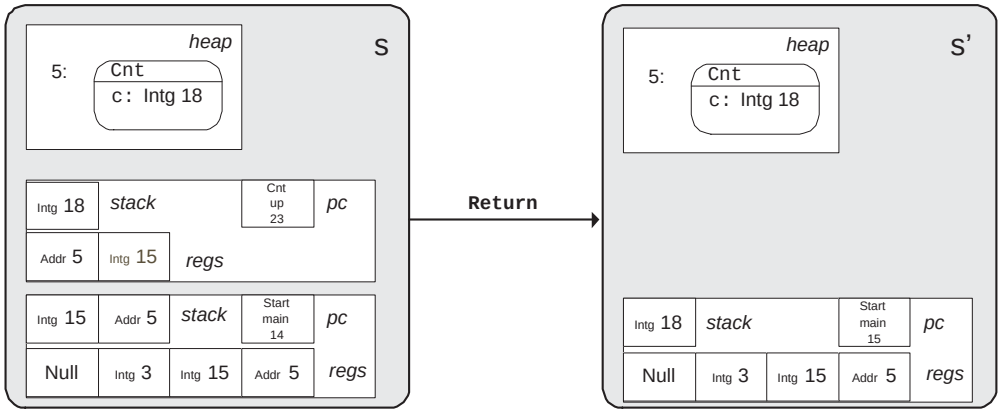


Figure 3.11: Return

ber d in the corresponding table entry (f,t,E,h,d) says how many elements are expected below the reference to the exception object. This feature is not required for programs translated from Java bytecode and complicates the definition of our abstract semantics *wpF* unnecessarily. For this reason we forbid it and define the wellformedness checker *wf* (see §6.2) such that it only accepts $d{=}0$ in all exception handler tables. The rule EXPT in Fig. 3.5 expects this and matches only if *find-handler* yields an operand stack containing just the reference, i.e. $[Addr\ xa]$. Another difference lies in the termination of both machines. The semantics $\xrightarrow{\text{jvm}}_1$ stops executing a program if it reaches a state with an empty frame stack. This can either be due to an uncaught exception or to a return from the main method. In other words, states with an empty frame stack are final states.

$final :: jvm\text{-}state \Rightarrow bool$
$final\ \sigma = snd\ (snd\ \sigma) = []$

The following lemma guarantees that $\xrightarrow{\text{jvm}}_1$ stops in case of a final state.

Lemma 3.1 $final\ \sigma \longrightarrow (\nexists \sigma'.\ P \vdash \sigma \xrightarrow{\text{jvm}}_1 \sigma')$

Our extended machine *effS* never reaches states with an empty frame stack. It stops immediately before such states would be reached. Both rules NRML and EXPT only permit progress to states with at least one frame. Having non empty frame stacks all the time simplifies the evaluation of some constructs in our assertion language, which we define in the next chapter. Apart from that both machines simulate each other for wellformed programs ($d = 0$).

Theorem 3.1 Every reachable, non-final transition in $\xrightarrow{\text{jvm}}_1$ also occurs in *effS*.

$$\frac{\begin{array}{l} wf\ \Pi \qquad C0 = fst\ (ipc\ \Pi) \qquad M0 = fst\ (snd\ (ipc\ \Pi)) \\ fst\ \Pi \vdash start\text{-}state\ (fst\ \Pi)\ C0\ M0 \xrightarrow{\text{jvm}} \sigma \\ fst\ \Pi \vdash \sigma \xrightarrow{\text{jvm}}_1 \sigma' \qquad \neg\ final\ \sigma' \end{array}}{\exists\, e'.\ ((pos\ \sigma,\ \sigma,\ e),\ pos\ \sigma',\ \sigma',\ e') \in effS\ \Pi}$$

Theorem 3.1 is extremely important to transfer our notion of safety, which we define relative to *effS*, to the actual behaviour a program shows on the non-extended Jinja VM $\xrightarrow{\text{jvm}}_1$. If *effS* could not simulate every step of $\xrightarrow{\text{jvm}}_1$, a program could be safe, just because we ignore some dangerous transitions of $\xrightarrow{\text{jvm}}_1$. From theorem 3.1 we know that a program than runs safely on *effS* also does so on $\xrightarrow{\text{jvm}}_1$. The theorem has a premise that σ is reachable and expresses it with $\xrightarrow{\text{jvm}}$, which is just a shortcut for the transitive reflexive closure of $\xrightarrow{\text{jvm}}_1$, i.e. $\xrightarrow{\text{jvm}} = (\xrightarrow{\text{jvm}}_1)^*$. We use this restriction to prove that both machines behave equally in case of exceptions. If we know that σ is reachable, we know that all positions recorded on the frame stack lie inside $domC\ (P,An)$. Then we know that all exception handler tables *find-handler* takes into account have been checked by *wf* and have $d = 0$.

The other way round, we also have a simulation, even for non-reachable states.

Theorem 3.2 Every transition in *effS* also occurs in $\xrightarrow{\text{jvm}}_1$.

$$\frac{wf\ (P,\ An) \qquad ((p,\ \sigma,\ e),\ p',\ \sigma',\ e') \in effS\ (P,\ An)}{P \vdash \sigma \xrightarrow{\text{jvm}}_1 \sigma'}$$

Theorem 3.2 states that the extended machine only shows behaviour that also occurs in $\xrightarrow{\text{jvm}}_1$. This means, the extended machine *effS* Π does not have unrealistic transitions, which would need to be verified although they never can occur in the non-extended one.

3.3 From Java to Jinja Bytecode

To be able to use our system on existing Java classfiles we developed a translation tool from Java classfiles to Jinja's class format. For this purpose `jissa` [1], a tool which disassembles Java classfiles to ASCII text, turns out to be handy. In contrast to other tools, e.g. `javap` or `BCEL` [2], `jissa` allows its users to customise the output easily. We translate all Java bytecode instructions to meaningful Jinja counterparts when possible.

Unsupported bytecode instructions are substituted by dummy instructions, e.g. `Goto` *1*, `Push` *X* or `Pop`. To get an idea of what is going on, let us look at an example. Compiling the Java program of Fig. 3.2 with `javac` version 1.4.1 and translating it with `j2jin` gives us the code shown in Fig. 3.3. Whenever a new object is created the `javac` compiler emits a `new C`, which allocates the object an pushes its reference onto the stack, followed by a constructor invocation. This is done by two instructions. First `dup` duplicates the reference, then `invokespecial` removes it and invokes the constructor. In Jinja, we do not support constructors. Objects are simply created with default field values. Instead of invoking constructors `j2jin` replaces `dup` with `Push` *Null* and `invokespecial` with `Pop`. In case all objects only have empty standard constructors, this amounts to the same. In Fig. 3.3 we have this situation at positions 5 and 6 of method *main*. Another deviation occurs when methods returning *Void* are invoked. In Jinja such methods yield *Unit* as result value, whereas real bytecode does not yield any result value. To deal with this, `j2jin` inserts a `Pop` instruction whenever a *Void* method returns. Before the `Return` it inserts `Push` *Unit*. For example at positions (*Start*,*main*,*11*) and (*Cnt*,*set*,*7*) we can observe this modification. Without these adjustments the Jinja bytecode verifier would reject the code. Since inserting instructions affects the control flow, `j2jin` also has to adjust jump targets and address labels in exception handler tables.

3.4 Conclusion

Many parts of the Jinja VM we presented in this chapter are taken from [55]. Since the concrete semantics is very helpful to understand the subtle transformations we will later on perform in the abstract semantics and because we added `IBin` and `IfIntCmp`, we have presented the full definition of *exec-instr*. We believe that the Jinja VM closely resembles the real Java VM as defined in [88]. The formalisation is mostly identical to the one Klein [54] used for the verification of the Java bytecode verifier. This formalisation has by now been completed to the full JVM. In case we are interested in extending our PCC framework towards full Java bytecode, this would be the machine to go for. Most features the Jinja VM misses should not cause conceptual challenges. Static methods are even simpler than dynamic ones. Non-standard constructors are challenging for the bytecode verifier [54] (field initialisation), but semantically they are just virtual method calls. Arrays require significant changes in the type system of the bytecode verifier and also in our assertion logic, which we introduce in the next chapter. However, Klein effectively shows in his verification of the bytecode verifier how to do it [54]. A big challenge are threads. Although we could simulate parallelism by making *effS* non-deterministic, this approach would lead to enormously big control flow graphs. This and the fact that one needs non-interferent annotations [9], make a naive approach practically infeasible.

4 Bytecode Assertion Logic

This chapter defines an assertion logic for Jinja bytecode. We need it to write annotations describing machine states, to express the safety policy as well as the verification conditions. The logic we propose here is first order and has interpreted symbols for integer arithmetic and virtual machine specific operations.

Over the years various logics for object oriented programs have been proposed. Abadi and Leino [5] define a Hoare Logic based on a combination of a type system and first order logic with equality. Oheimb [75] uses a shallow embedding of Isabelle/HOL to define a Hoare logic for Java. A very prominent annotation language for Java is JML [58] or the down-sized version of it used in ESC Java [40]. However, all of the logics have been designed for source languages, not for bytecode. To verify bytecode people suggested other ways. For example the LOOP tool [91] and the JIVE verifier [62] can be used to turn Java classes into proof obligations in form of theory files for a theorem prover. Since these tools are complex, but not verified themselves, they face the same problems with trustworthiness as Touchstone PCC (see §1.3.1). Strother Moore has formalised the Java VM in ACL2 and proposes to verify bytecode programs directly on the semantics [3]. He thus follows FPCC (see §1.3.2) in its purest form. Recently, also the interest in *logics* for bytecode started growing [79, 14, 10]. However, as we point out in the conclusion, these publications do not focus on the assertion level. Most try to port Hoare Logic to the bytecode level in order to reason about input/output relations for terminating bytecode. We are rather interested in safety and intermediate states. Hence, our main interest is the assertion language.

Albeit the data abstractions of Java and its bytecode are similar, there are differences in the states. The states of a Java VM contain a lot more details than those at the source level.

For example, the operand stack is always empty when a Java statement is fully processed. An assertion logic for Java does therefore not have to bother about the operand stack. At the bytecode level we want to able to reason about this stack. We want to simulate all the fine grained effects bytecode instructions have on it at a syntactic level.

Since safety violations not just happen at the points when Java statements begin or end, our assertion logic needs to be able to take snapshots of the VM before and after the

execution of single instructions. The assertion logic we present in this chapter is tailored for this purpose. As we will prove in §6.6 it can express weakest preconditions for every Jinja instruction and any postcondition.

4.1 Syntax and Semantics of Assertions

Our assertion language has the syntax shown in Fig. 4.1. It is first order arithmetic with special constructs for adequate modelling of virtual machine states. All these constructs are used to express our safety policy (no arithmetic overflow) and weakest preconditions for Jinja VM instructions. Next, we explain the semantics of all language constructs.

datatype *expr* =

Rg nat	register
| *St nat*	operand stack cell
| ⌊*val*⌋	constant value
| *NewA nat*	address for nth new object
| *Gf vname cname expr*	get field value
| *Ty expr ty*	type check
| *FrNr*	height of method frame stack
| *Pos pos*	position check
| *Call expr*	evaluation in call state
| *Catch cname expr*	evaluation in catch state
| *Num expr num-op expr*	numerical expression
| *Rel expr rel-op expr*	numerical relation
| ⌊*if expr* ⌊*then*⌋ *expr* ⌊*else*⌋ *expr*⌋	conditional
| *expr* ⌊=⌋ *expr*	equality
| ⌊¬⌋ *expr*	negation
| *expr* ⌊⇒⌋ *expr*	implication
| ⌊⋀⌋ *expr list*	conjunction
| *Lv nat*	logical variable
| ⌊∀⌋ *nat. expr*	quantification (logical vars)

Figure 4.1: Jinja bytecode assertion language.

Each expression can be evaluated for a given *jbc-state* and yields some Jinja value.

evalE :: *jbc-prog* ⇒ *jbc-state* ⇒ *expr* ⇒ *val option*

The expressions come in two categories. From *Rg nat* to *Catch expr* we have JVM specific constructs. These are needed to access various parts of Jinja states in annotations. The remaining constructs are purely logical and are required to construct verification conditions or to express safety policies.

4.1.1 JVM Constructs

Since we want to use the assertion logic to abstract Jinja states, we need various constructs to access different parts of such states. Most instructions manipulate only the topmost method frame. Instead of making the whole frame stack accessible in the logic, which would complicate the evaluation range, we decided to use constructs for individual parts only. In the following we define the meaning of each expression. The reader may check these definitions against the example expressions shown in Fig. 4.3, which we evaluate under the situation depicted in Fig. 4.2. This situation occurs if we start the program from Fig. 3.3 with `x0` set to -3, instead of 3.

With *Rg k* and *St k* we access the kth register or element on the operand stack. If k exceeds the register range or stack height, these expressions yield *None*.

$$\begin{aligned} \mathit{evalE}\ \Pi\ (p,\sigma,e)\ (\mathit{Rg}\ k) = (&\mathsf{let}\ (x,h,\mathit{frs})=\sigma;\ (\mathit{st},\mathit{rg},p)=\mathit{hd}\ \mathit{frs}\\ &\mathsf{in}\ (\mathsf{if}\ k < \mathit{length}\ \mathit{rg}\ \mathsf{then}\ \lfloor \mathit{rg}_{[k]} \rfloor\\ &\quad \mathsf{else}\ \mathit{None})) \end{aligned}$$

$$\begin{aligned} \mathit{evalE}\ \Pi\ (p,\sigma,e)\ (\mathit{St}\ k) = (&\mathsf{let}\ (x,h,\mathit{frs})=\sigma;\ (\mathit{st},\mathit{rg},p)=\mathit{hd}\ \mathit{frs}\\ &\mathsf{in}\ (\mathsf{if}\ k < \mathit{length}\ \mathit{st}\ \mathsf{then}\ \lfloor \mathit{st}_{[k]} \rfloor\\ &\quad \mathsf{else}\ \mathit{None})) \end{aligned}$$

Constants $\lfloor v \rfloor$ evaluate to their values v.

$$\mathit{evalE}\ \Pi\ s\ \lfloor v \rfloor = \lfloor v \rfloor$$

To improve readability we sometimes abbreviate constants like $\lfloor \mathit{Intg}\ 5 \rfloor$, $\lfloor \mathit{Bool}\ \mathit{True} \rfloor$ or $\lfloor \mathit{Bool}\ \mathit{False} \rfloor$ with $\lfloor 5 \rfloor$, $\lfloor T \rfloor$ and $\lfloor F \rfloor$.

The *NewA n* expression yields the reference that is allocated in the heap for the nth object. If the heap is full, it yields *Null*. We use this operator to express the semantics of the *New* instruction. This is one of the few situations where weakest precondition construction becomes difficult. The new state has something, e.g. a new address, that is not directly accessible in the current state. In many cases the *NewA* expression can be avoided as [22] shows. In our *wpF* operation, we will also replace field accesses of newly created objects by their default values.

Eliminating all instances of *NewA* could be achieved at the level of formulas [22], which in our cases are expressions always yielding values of the form *Bool b*. However, using this elimination right from the start leads to difficulties. Our proofs for the *wpF* function would become a lot more involved, as non-trivial transformations and the distinction between expression and formulas lead to complications in the structural induction scheme. Eliminating *NewA* where possible in a post-processing optimiser is an alternative we prefer. Moreover, *NewA* turns out to be useful when it comes to exceptions. In our

completeness proof for the VCG, we require the assertion language to be expressive enough for sharp branch conditions of all instructions. In case of the *New* instruction, we must be able to express the situation when an *OutOfMemory* exception occurs. The expression $NewA\ 0\ \llbracket\rrbracket\ \lfloor Null \rfloor$ exactly does that. To describe the semantics of *NewA n* formally, we use the auxiliary function *new-Addr h* (see [55]), which yields either $\lfloor a \rfloor$ if a is the reference that is allocated next, or *None* if the heap is full. With *evalNewA* we repeat allocating addresses until we get the address for the nth new object.

$evalE\ \Pi\ s\ (NewA\ n) = (\mathsf{let}\ (p,\ \sigma,\ e) = s;\ (x,\ h,\ frs) = \sigma\ \mathsf{in}\ \lfloor evalNewA\ h\ n \rfloor)$
$evalNewA\ h\ 0 = (\mathsf{case}\ new\text{-}Addr\ h\ \mathsf{of}\ None \Rightarrow Null \mid \lfloor a \rfloor \Rightarrow Addr\ a)$
$evalNewA\ h\ (Suc\ n) = evalNewA\ (h(the\ (new\text{-}Addr\ h) \mapsto arbitrary))\ n$

To evaluate *Gf F C ex*, which corresponds to $(C)ex.F$ in Java, we first check whether *ex* evaluates to an address value. If so, we fetch the value of the corresponding object field, otherwise we return *None*.

```
evalE Π s (Gf F C ex) =
(case evalE Π s ex of None ⇒ None
 | ⌊v⌋ ⇒
    case v of
    Addr a ⇒
      ⌊let (p, σ, e) = s; (x, h, frs) = σ; (D, fs) = the (h a)
       in the (fs (F, C))⌋
    | - ⇒ None)
```

To check the exact type of some expression, we use *Ty ex tp*. Note that this check does not take the class hierarchy into account. Subtyping can be expressed by a disjunction of *Ty ex tp* expressions.

```
evalE Π s (Ty ex tp) =
⌊Bool
  (case evalE Π s ex of None ⇒ False
   | ⌊v⌋ ⇒
      case v of Unit ⇒ tp = Void
      | Null ⇒ tp = NT
      | Bool b ⇒ tp = Boolean
      | Intg i ⇒ tp = Integer
      | Addr a ⇒
          let (p, σ, e) = s; (x, hp, frs) = σ
          in case hp a of None ⇒ False
             | ⌊ob⌋ ⇒ tp = Class (fst ob))⌋
```

Expression *FrNr* counts the number of frames on the frame stack. This is important to

specify initial states with *initF* correctly.

$evalE\ \Pi\ s\ FrNr = (\mathsf{let}\ (p,\ \sigma,\ e) = s;\ (x,\ h,\ frs) = \sigma\ \mathsf{in}\ \lfloor Intg\ (int\ |frs|)\rfloor)$

To check the current program position we have *Pos p*. In addition this expression also checks other properties that should hold whenever a program Π reaches p at runtime. The frame stack must not be empty and the position recorded on the topmost frame must coincide with the program counter p, the first component of a *jbc-state*. In addition, we check the exception flag to be unset. Note that *effS* uses this flag only internally while it searches for a handler, but never yields a successor state with a set exception flag. Finally, we require the frame stack to be wellformed, i.e. *wf-frs* Π p.

$$
\begin{array}{l}
evalE\ \Pi\ s\ (Pos\ q) = \\
\lfloor Bool \\
\quad (\mathsf{let}\ (p,\ \sigma,\ e) = s;\ (x,\ h,\ frs) = \sigma;\ (st,\ rg,\ p_f) = hd\ frs \\
\quad\ \mathsf{in}\ q = p \wedge p_f = p \wedge x = None \wedge frs \neq [] \wedge wf\text{-}frs\ \Pi\ frs)\rfloor
\end{array}
$$

The predicate *wf-frs* Π *frs* checks two things: First, the instruction of the bottom frame lies in the main method, whose class and method name is determined by *ipc* Π. Second, the program position of each frame lies in the code domain and points to an instruction that invokes the method of the frame above. We write *callers* Π p for a list of all positions that call the method p belongs to.

$$
\begin{array}{l}
callers :: jbc\text{-}prog \Rightarrow pos \Rightarrow pos\ list \\
callers\ \Pi\ (C,\ M,\ pc) = [p_c{\in}domC\ \Pi\ .\ \exists\, n.\ cmd\ \Pi\ p_c = \lfloor \texttt{Invoke}\ M\ n\rfloor] \\
wf\text{-}frs :: jbc\text{-}prog \Rightarrow (val\ list \times val\ list \times pos)\ list \Rightarrow bool \\
wf\text{-}frs\ \Pi\ frs = \\
\forall\, i{<}|frs|. \\
\quad \mathsf{let}\ (st_i,\ rg_i,\ p_i) = frs_{[i]};\ (st_c,\ rg_c,\ p_c) = frs_{[i\ +\ 1]}; \\
\qquad (C_i,\ M_i,\ pc_i) = p_i;\ (C_0,\ M_0,\ pc_0) = ipc\ \Pi \\
\quad \mathsf{in}\ p_i \in set\ (domC\ \Pi) \wedge \\
\qquad (\mathsf{if}\ i\ +\ 1 = |frs|\ \mathsf{then}\ C_i = C_0 \wedge M_i = M_0\ \mathsf{else}\ p_c \in set\ (callers\ \Pi\ p_i))
\end{array}
$$

With *Call* and *Catch* we evaluate expressions in previous states. The *Call ex* expression evaluates *ex* in the call state of the current method. This helps to specify postconditions of methods modularly. For example, annotating a `Return` instruction with $Rg\ 1 \sqsubseteq Call\ (Rg\ 1) \sqsupseteq \underline{1}$ means that the returning method has incremented register *1*.

This technique is related to primed variables in VDM [53], except that we can set entire expressions into a different temporal context, just like temporal logic operators do. This is important, as some postconditions need old values of object fields or other parts of the heap. Another reason is that we use this operator to restore the call context, when we

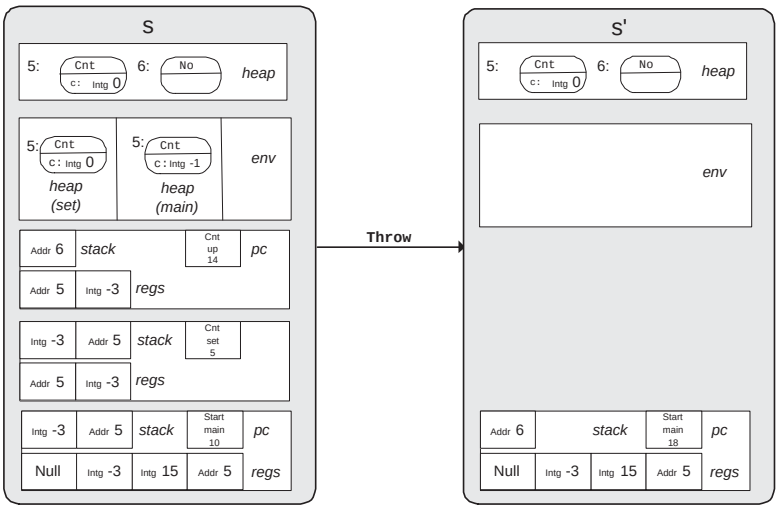


Figure 4.2: Throw

i	Q_i	$evalE\ \Pi\ s\ Q_i$
1	*Rg 1*	$\lfloor Intg\ -3 \rfloor$
2	*St 0*	$\lfloor Addr\ 6 \rfloor$
3	*NewA 0*	$\lfloor Addr\ 7 \rfloor$
4	*Pos (Cnt,up,14)*	$\lfloor Bool\ True \rfloor$
5	*Call (Rg 0)*	$\lfloor Addr\ 5 \rfloor$
6	*Catch No (Rg 0)*	$\lfloor Null \rfloor$
7	*Catch No (Gf c Cnt (Rg 0))*	$\lfloor Intg\ -1 \rfloor$

Figure 4.3: Example expressions and evaluations

compute the verification condition of a method return. For example, if register *1* had value *Intg 5* before the method call, the programmer might annotate the call position with $Rg\ 1 \sqsupseteq \underline{5}$ and the return position with $Rg\ 1 \sqsupseteq \underline{6}$. Our VCG would then produce the following proof obligation, where we have to show that the postcondition together with the call annotation (evaluated in the call state!) imply the annotation at the return position:
$(Rg\ 1 \sqsupseteq Call\ (Rg\ 1) \sqinter \underline{1} \sqinter Call\ (Rg\ 1 \sqsupseteq \underline{5})) \sqsupseteq\!\!\!\rightarrow Rg\ 1 \sqsupseteq \underline{6}$. Details about this construction can be found in [98] and in §6.5. We use the auxiliary function *callstate* to restore the call state of the current method. The program counter, registers and operand stack at call time are taken from the method frame beneath. The heap can be restored from the recordings in the environment. In case of a `main` method state (no caller), *Call ex* evaluates to *None*.

$evalE\ \Pi\ s\ (Call\ ex) =$
$(\mathsf{let}\ (p,\ \sigma,\ e) = s;\ (x,\ h,\ frs) = \sigma$
$\ \mathsf{in\ if}\ |frs| \leq 1\ \mathsf{then}\ None\ \mathsf{else}\ evalE\ \Pi\ (callstate\ s)\ ex)$

$callstate{::}jbc\text{-}state \Rightarrow jbc\text{-}state$
$callstate\ (p,(x,h,f \cdot (s,r,p') \cdot frs),e) = (p',(None,hd\ (cs\ e),(s,r,p') \cdot frs),e(\!|cs{:=}tl(cs\ e)|\!))$
$callstate\ s = s$

Exceptions impose a similar problem than method returns. In case the current method provides a handler, exceptions are not a big deal. Control is transferred to the handler and the operand stack becomes emptied. If there is no handler, frames are chopped off the frame stack until a method with a catching handler is found. The effect on the state is quite drastic and imposes a real problem for our weakest precondition operator, which has to reflect such changes by transforming expressions. How many frames exception handling climbs up is hard to determine statically. For example, if the current method is recursive and only the top level method has a catching handler, the number of removed frames coincides with the current recursion depth. Our solution to this problem is to introduce a special operator for exception handling. Just like *Call ex* the construct *Catch X ex* evaluates *ex* in a previous state. In this case we restore the state under which we have last been in the `try` block that has a catching handler for exception *X*. The auxiliary function *catchstate* chops off frames until a catching handler is found. For this purpose it uses *ex-table P C M* and *match-ex-table P X pc xt* (see [55]). The first fetches a method's exception table, while the second searches such a table for an entry that catches exceptions of type *X* thrown from *pc*. It yields a pair *pd* consisting of the entry position *pch* of the catching handler and a number *d*, i.e. $pd = \lfloor(pch,d)\rfloor$, or *None* if there is no such handler. Once a catching handler is found, *catchstate* restores the state under which we have last been in the catching method. The frame stack of this state is a suffix of the current frame stack. The heap at that time is recorded in the environment of the current state. Note that the resulting state is not the state under

which the handler is entered, but the state under which we have last been in the method with the `try` block. For the former state we would use *find-handler*, which in contrast to *catchstate* keeps the current heap and transfers control into the handler.

$$
\begin{array}{l}
\mathit{evalE}\ \Pi\ s\ (\mathit{Catch}\ X\ ex) = \\
(\mathsf{let}\ (p,\ \sigma,\ e) = s;\ (x,\ h,\ \mathit{frs}) = \sigma \\
\ \mathsf{in}\ \mathsf{if}\ |\mathit{frs}| \leq 1\ \mathsf{then}\ \mathit{None}\ \mathsf{else}\ \mathit{evalE}\ \Pi\ (\mathit{catchstate}\ (\mathit{fst}\ \Pi,\ X,\ s))\ ex)
\end{array}
$$

$$
\begin{array}{l}
\mathit{catchstate} :: (\mathit{jvm\text{-}prog} \times \mathit{cname} \times \mathit{jbc\text{-}state}) \Rightarrow \mathit{jbc\text{-}state} \\
\mathit{catchstate}\ (P,X,(p,(x,h,\mathit{fr} \cdot (st,rg,p) \cdot \mathit{frs}),e)) = \\
(\mathsf{let}\ (C,M,pc){=}p\ \mathsf{in}\ (\mathsf{case}\ (\mathit{match\text{-}ex\text{-}table}\ P\ X\ pc\ (\mathit{ex\text{-}table\text{-}of}\ P\ C\ M)) \\
\ \ \mathit{of}\ \mathit{None} \Rightarrow \mathit{catchstate}\ (P,X,(p,(\mathit{None},\mathit{hd}\ (cs\ e),(st,rg,p) \cdot \mathit{frs}),e(\!|cs{:=}tl(cs\ e)|\!))) \\
\ \ |\ \lfloor pd \rfloor \Rightarrow (p,(\mathit{None},\mathit{hd}\ (cs\ e),(st,rg,p) \cdot \mathit{frs}),e(\!|cs{:=}tl\ (cs\ e)|\!)))) \\
\mathit{catchstate}\ (P,X,s) = s
\end{array}
$$

4.1.2 Logical Constructs

The arithmetic, relational, conditional and logical expressions are evaluated recursively. First, we evaluate the argument expressions, then we apply the corresponding arithmetic, relational, conditional or logical operator on the results. This is done by the lifting functions *liftI* and *liftR*, which we introduced in §3.1. If any argument value has not the expected type the result becomes *None*. For better readability, we write concrete instances of numerical expressions $\mathit{Num}\ e\ \mathit{no}\ e'$ and relations $\mathit{Rel}\ e\ \mathit{ro}\ e'$ in the following style: $e \dot{+} e'$, $e \dot{-} e'$, $e \dot{*} e'$, $e \dot{<} e'$, $e \dot{\leq} e'$, $e \dot{>} e'$, $e \dot{\geq} e'$ and $e \dot{=} e'$.

$$\mathit{evalE}\ \Pi\ s\ (\mathit{Num}\ e1\ \mathit{no}\ e2) = \mathit{liftI}\ (\mathit{numop}\ \mathit{no},\ \mathit{evalE}\ \Pi\ s\ e1,\ \mathit{evalE}\ \Pi\ s\ e2)$$

$$\mathit{evalE}\ \Pi\ s\ (\mathit{Rel}\ e1\ \mathit{ro}\ e2) = \mathit{liftR}\ (\mathit{relop}\ \mathit{ro},\ \mathit{evalE}\ \Pi\ s\ e1,\ \mathit{evalE}\ \Pi\ s\ e2)$$

$$
\begin{array}{l}
\mathit{evalE}\ \Pi\ s\ (\dot{\mathit{if}}\ b\ \dot{\mathit{then}}\ t\ \dot{\mathit{else}}\ e) = \\
(\mathsf{if}\ \mathit{the\text{-}Bool}\ (\mathit{evalE}\ \Pi\ s\ b)\ \mathsf{then}\ \mathit{evalE}\ \Pi\ s\ t\ \mathsf{else}\ \mathit{evalE}\ \Pi\ s\ e)
\end{array}
$$

$$\mathit{evalE}\ \Pi\ s\ (e1 \dot{=} e2) = \lfloor \mathit{Bool}\ (\mathit{evalE}\ \Pi\ s\ e1 = \mathit{evalE}\ \Pi\ s\ e2) \rfloor$$

$$\mathit{evalE}\ \Pi\ s\ (\dot{\neg}\ ex) = \lfloor \mathit{Bool}\ (\neg\ \mathit{the\text{-}Bool}\ (\mathit{evalE}\ \Pi\ s\ ex)) \rfloor$$

$$\mathit{evalE}\ \Pi\ s\ (e1 \dot{\Rightarrow} e2) = \lfloor \mathit{Bool}\ (\mathit{the\text{-}Bool}\ (\mathit{evalE}\ \Pi\ s\ e1) \longrightarrow \mathit{the\text{-}Bool}\ (\mathit{evalE}\ \Pi\ s\ e2)) \rfloor$$

$evalE\ \Pi\ s\ (\bigwedge\!\!\!\bigwedge\ es) = \lfloor Bool\ (\forall\, ex{\in}set\ es.\ the\text{-}Bool\ (evalE\ \Pi\ s\ ex))\rfloor$

A logical expression ex is true if it evaluates to $\lfloor Bool\ True\rfloor$, otherwise it is false. From Winskel [99] we take the idea of distinguishing program and logical variables. The first (registers, stack ...) depend on the *jvm-state* and may be modified by instructions. The latter are evaluated in a separate binding $lv\ e$, which we made part of the environment e in *jbc-state*, and are unaffected by instructions. In the substitutions we use to express the effect of instructions, we will neither transform nor introduce logical variables. Hence, no renaming of bound variables is required. From our experience with a different instantiation, where we only had one notion of variable, we can say that this greatly simplifies the *wpF* operator.

$evalE\ \Pi\ (p,\ \sigma,\ e)\ (Lv\ k) = lv\ e\ k$

Quantification only binds logical variables. The expression $\forall\!\!\!\forall\ x.\ ex$ holds, if ex holds no matter what value v the logical variable $Lv\ x$ is bound to.

$evalE\ \Pi\ (p,\ \sigma,\ e)\ (\forall\!\!\!\forall\ x\ ex) =$

$\lfloor Bool\ (\forall\, v.\ the\text{-}Bool\ (evalE\ \Pi\ (p,\ \sigma,\ e(\!|lv := (lv\ e)(x := v)|\!))\ ex))\rfloor$

4.2 Logical Judgments

To use this expression language as a logic, we need judgements for validity and provability. Models are program states under which an expression evaluates to $\lfloor Bool\ True\rfloor$.

$\Pi{,}s \models\ ex = evalE\ \Pi\ s\ ex = \lfloor Bool\ True\rfloor$

Provability $\vdash$ is usually defined by giving a set of axioms and inference rules. However, we can also "define" provability semantically and use the inference system of HOL for proofs. We regard a formula as provable if we can prove in HOL that it holds for all states.

$\Pi \vdash ex = (\forall\, s.\ \Pi{,}s \models\ ex)$

In earlier instantiations, we also defined $\vdash$ semantically, but only required all reachable states [97] or even only anno-reachable states [98] to satisfy the expression. Although this works for our purpose of proving safety, it is rather unnatural as it drags the operational semantics into the logic. The notion of provability as defined above agrees with the semantical conception of provable formulas in standard text books on logic[46] and is completely detached from the program semantics. This way the safety logic abstracts

from the programming language and proving verification conditions becomes a purely logical task.

4.3 Design Choices

The assertion language we just introduced is the result of various considerations. In this section we shed some light on why we ended up with this format.

4.3.1 Deep or Shallow?

When one embeds a language into a theorem prover a major decision is whether to make it shallow or deep. In this chapter we have realised a deep embedding, because we defined expressions as a datatype *expr*. For a simpler instantiation of our PCC framework, we formalised safety logics in both styles and compared them [96]. In a *shallow embedding* one reuses the syntax of the theorem prover and defines language constructs purely semantically [73, 75, 83]. In our case this means expressions are modelled as HOL predicates on Jinja states, i.e. $expr = (jbc\text{-}state \Rightarrow bool)$. A shallow embedding has many advantages. First of all, it only costs little effort to formalise. Second, if the logic of the theorem prover is very expressive, which clearly is the case for HOL, we can easily describe complex sets of states. We can define functions in HOL and use them in assertions. To transform the meaning of a shallow embedded assertion we can compose it witch a function that modifies the state. For example Hoare's assignment rule in a shallow embedding looks like $\{\lambda\ s.\ Q\ s(x{:=}e\ s)\}\ x{:=}e\ \{\ Q\ \}$. Instead of syntactic substitutions, we update the semantical object, the state. Avoiding substitutions clearly saves proving effort. The downside is that, we cannot talk in the theorem prover about the structure of language constructs. All we can do is application. If we want to compare two shallow embedded expressions, e.g. $e = e'$, we end up with comparing two functions semantically, which is undecidable in general. Also induction on the size of expressions is impossible. In a *deep embedding* one defines a datatype for the language, and thus introduces syntax. This allows to do induction, comparison, and optimisation. The latter two are in particular important to our application. With comparison we can extract information from language expressions. In our case the successor function *succsF* will make use of this and extract information about the potential successors of method invocations and exception throws from the annotations at these instructions. Since the syntax is visible, we can also do optimisations of formulas. In case of verification conditions this turned out to greatly reduce the size of formulas and their proofs. Even without optimisations the proofs of verification conditions in the deep embedding turned out to be smaller. The reason is, that in the deep embedding semantic transformations

are done by substitutions, which take place inside the VCG. When we prove the resulting formula this work is already done. In the shallow embedding the VCG expresses semantic transformations via λ terms that change the state. These transformations are carried out when we do β contraction. In contrast to syntactic substitutions this happens inside the proof, when we apply the semantical judgement to the verification condition.

4.3.2 What Language Constructs?

In our assertion language one finds three categories of expressions reflecting the three purposes of the language. We have purely logical expressions for constructing verification conditions. Then we have arithmetic expressions, which are important to specify a safety policy against arithmetic overflow. Finally, we have various constructs for accessing different parts of Jinja states in order to abstract them in annotations. All these constructs evaluate to primitive Jinja values. This makes the language first order and expressions easily composable. When we introduced our language constructs we found that one cannot blindly add new constructs depending on the state. If we introduce state dependent language construct, e.g. *Gf F C*, we have to make sure that we can also express all effects instructions have on it. For example, once we introduced *Gf*, we realised that we also need *NewA* and $\llbracket$*if* - $\llbracket$*then*$\rrbracket$ - $\llbracket$*else*$\rrbracket$ to compute weakest preconditions. If we dropped all heap dependent language constructs, we could also achieve an expressive assertion logic, because no heap transformation would affect any logical construct. An extreme example of an expressive assertion logic would be a logic with only one formula, e.g. $\llbracket T \rrbracket$. It is expressive, because all weakest preconditions of assertions would be $\llbracket T \rrbracket$, which is representable in the logic.

4.3.3 Typed or Untyped?

For simplicity, our assertion language is untyped. We do not even distinguish between formulas and expressions. Earlier instantiations [96] showed that this distinction leads to duplication of functions and lemmas for both types. The uniform representation avoids this and still allows categorisation with type checking functions. In our case the semantics of ill-typed expressions is *None*, which is one of the many representations of the truth value *False*.

4.3.4 Higher Order Abstract Syntax

A convenient way to reduce the number of different constructs would have been to define *expr* as a datatype with higher order abstract syntax. That is using constructors

with functions as arguments. For example one could define a binary *Apply* operator as follows:

$expr = \ldots \mid Apply\ (val \Rightarrow val \Rightarrow val)\ expr\ expr$

In the semantics one would then simply apply the provided function on the results for the argument expressions.

$evalE\ \Pi\ s\ (Apply\ f\ e1\ e2) = (lift\ f)\ (evalE\ \Pi\ e1)\ (evalE\ \Pi\ e2)$.

Many expressions like relations, binary arithmetic and so on could be defined in that way. Since the function f only operates on values (not states!) the definition of a weakest precondition operator would also cause no problems. Only the argument expressions need to be modified. However, such a representation of formulas also introduces problems. First, like in a shallow embedding the syntax of the operator f is not accessible for other HOL functions. This hampers extracting information from or optimising such formulas. For example the formula $Apply\ (\lambda\ x\ y.\ x + y)\ \underline{2}\ \underline{3}$ cannot be optimised to $\underline{5}$. Second, the notation is bound to Isabelle/HOL. Employing other provers for the safety logic becomes quite difficult. Third, the representation of such formulas causes problems for the code generator. If we generate code for the expression above the generator would turn $\lambda\ x\ y.\ x + y$ into ML code computing + rather than leaving the representation as it is. Although the code generator now supports a Quoting mechanism that allows to handle this, it is more convenient to avoid it.

4.3.5 Inference Rules?

Usually one defines provability $\vdash$ with axioms and inference rules. For example:

$\Pi \vdash \neg\ (\neg\ ex)\ \Longrightarrow\ ex$

In this fashion we could define a calculus for our safety logic and then connect provability $\vdash$ with validity $\models$ by conducting proofs for correctness and completeness. If one wants to employ a very specialised safety logic, such as a type system that ensures some kind of type safety, this would be the way to go. In our case the situation is different. Except for the Jinja specific operations our safety logic is what the literature calls first order arithmetic (FOA). We can quantify over primitive values and have arithmetic operations with fixed interpretation. Since rules for such a standard logic are well researched we decided against a deep embedded inference system. Another reason is that rules of the form above, do not work well with the Isabelle simplifier, which rewrites $=$, but not $\Longrightarrow$. The equation $\neg\ (\neg\ ex) = ex$ would work well as a rewrite rule, but is logically wrong, because both sides are syntactically different elements of the datatype *expr* and $=$ means syntactic equivalence. For these reasons we define $\vdash$ directly with $\models$ and prove formulas on the semantic side. This way we can employ Isabelle/HOL inference rules

$$\frac{\Pi,s \models\ f1 \qquad \Pi,s \models\ f2}{\Pi,s \models\ \lfloor\bigwedge\rfloor\ [f1,\ f2]}\textsc{ConjI}$$

$$\frac{\forall\, v.\ \Pi,(p,\ \sigma,\ e(\!|lv := (lv\ e)(x := v)|\!)) \models\ f}{\Pi,(p,\ \sigma,\ e) \models\ \lfloor\forall\rfloor\ x\ f}\textsc{AllI}$$

$$\frac{\exists\, n.\ \Pi,s \models\ ex \sqsupset \lfloor Intg\ n\rfloor}{\Pi,s \models\ Ty\ ex\ Integer}\textsc{TyIntI}$$

$$\frac{\Pi,s \models\ \lfloor Intg\ 1\rfloor \lesssim FrNr \qquad \Pi,s \models\ \sqsubset\ (Call\ f)}{\Pi,s \models\ Call\ (\sqsubset\ f)}\textsc{CallNeg}$$

Figure 4.4: Semantical proof rules

and decision procedures for our safety proofs. We can also derive rules to guide the proof construction. For example Fig. 4.4 shows such derived rules.

The rules ConjI and AllI connect the logical symbols of the assertion logic to the corresponding operators in Isabelle/HOL. The rule CallNeg is just one example of many other rules we have to push *Call* or *Catch* operators inside composed expressions. Note that this rule requires the frame stack to have at least two entries. The rule TyIntI resolves type expressions to Isabelle variables of the corresponding type. When the existential quantifier becomes removed, which is typically done automatically by Isabelle's proof procedures, one can use the resulting equation to simplify other expressions. For example the following lemma can be proven automatically this way.

lemma $\Pi,s \models (Ty\ ex\ Integer \Rightarrow (ex \lesssim (ex \dotplus \lfloor Intg\ 1\rfloor)))$
by *clarsimp*

To achieve this, we actually do not use the rules in the format shown above, but in form of equations on the *evalE* function. For example the rule TyIntI can be turned into the following rewrite rule:

$evalE\ \Pi\ s\ (Ty\ ex\ Integer) = \lfloor Bool\ (\exists\, x.\ evalE\ \Pi\ s\ ex = \lfloor Intg\ x\rfloor)\rfloor$

4.4 Conclusion

In its current form our assertion logic is biased to verify absence of arithmetic overflows. Other safety policies, in particular those involving complex statements about the heap

structure are not supported. Properties like referential reachability require higher order concepts such as transitive closure. To support this, we plan to extend our logic with primitive recursion. This way we can also integrate other analysers, e.g. [59, 87, 85].
The literature also offers alternatives to verify bytecode.
Quigley [79] formalises a Hoare Calculus for bytecode in Isabelle/HOL using a shallow embedded assertion language. The focus of this work is to state and prove sound a rule for bytecode patterns resembling a WHILE loop. A glimpse at the control flow graph we show in Fig. 5.1 reveals that bytecode is naturally flat and unstructured. In particular exceptions mess up the control flow.
Nevertheless, Hoare rules can be ported to that level. Clint and Hoare [32] propose to view jumps to a label l as calls to an embedded procedure l, whose body starts with l and ends where instructions exit the block. The benefit of porting Hoare rules to bytecode is that one can translate high level proofs to the target language. For example the bytecode logic proposed in [10], which is also formalised in Isabelle/HOL as a shallow embedding, is designed to replay the type inferences of a high level type system for memory consumption [52] (see §1.3.5). The bytecode Hoare Logic proposed in [14] is also accompanied by a compiler that translates high level Hoare proofs.
In our case we keep the VCG generic and cannot directly employ rules designed for the structure of a particular programming language. However, a way to support source level reasoning is to translate annotations. Our VCG emits similar proof obligations as Hoare rules have in their side conditions. Hence, we think that primarily the complexity of annotations determines how hard a program is to verify. Annotations are directly influenced by a program's semantics and not so much by its structure. Necula [68] backs up this claim with his observations on how little code optimisations affect the provability of verification conditions. Hence, if we can translate invariants, pre- and postconditions form the source level, verifying a program at the bytecode level should cause similar difficulties than on the source level. Pavlova [76] proposes to translate JML annotations into BCSL, the so-called bytecode specification language. The low level annotations can then be given to the JACK tool [25], which emits verification conditions for an external theorem prover, such as Coq, Simplify or PVS.
The BCSL language is also defined with a specific syntax (an extension of JML). However, JML is quite complex as a logic. For example, it allows to evaluate Java methods inside logical expressions. Hence, it remains to be seen, if it can be sufficiently supported by theorem provers. Our assertion logic is not that expressive, but since we use Isabelle/HOL inference rules to conduct safety proofs, we can be sure that it is sound.

5 Control Flow and Abstract Semantics

Our verification conditions are structured according to a program's control flow. We instantiate a function that determines statically what successors a Jinja instruction can have and under which situations these are accessible. Each control flow edge abstracts an instruction's effect on the state. Our abstract semantics mimics this effect by transforming formulas such that their meaning is preserved.

In this chapter we instantiate the essential parameters of our VCG, the control flow function *succsF* and the abstract semantics in form of *wpF*, a weakest precondition operator, and *initF*, a formula specifying initial states.

5.1 Control Flow Approximation

Our verification conditions are structured after the control flow graph, which consists of positions and edges. For example Fig. 5.1 shows the control flow for the counter application from Fig. 3.3. One can observe that many edges lead to instruction number *23* of the *main* method. This is where the exception handler for general exceptions starts. Some instructions even have more than one edge leading to a handler. For example at node *14* of the *up* method we have a `Throw` instruction with two edges leading to the handler at position *18* and two leading to the one at position *23*. This is because method *up* has two call contexts, namely from method *main* and *set*, at each of which either a *NullPointer* or a *No* exception may be thrown. For each of these 4 situations we have to construct a different branch condition.

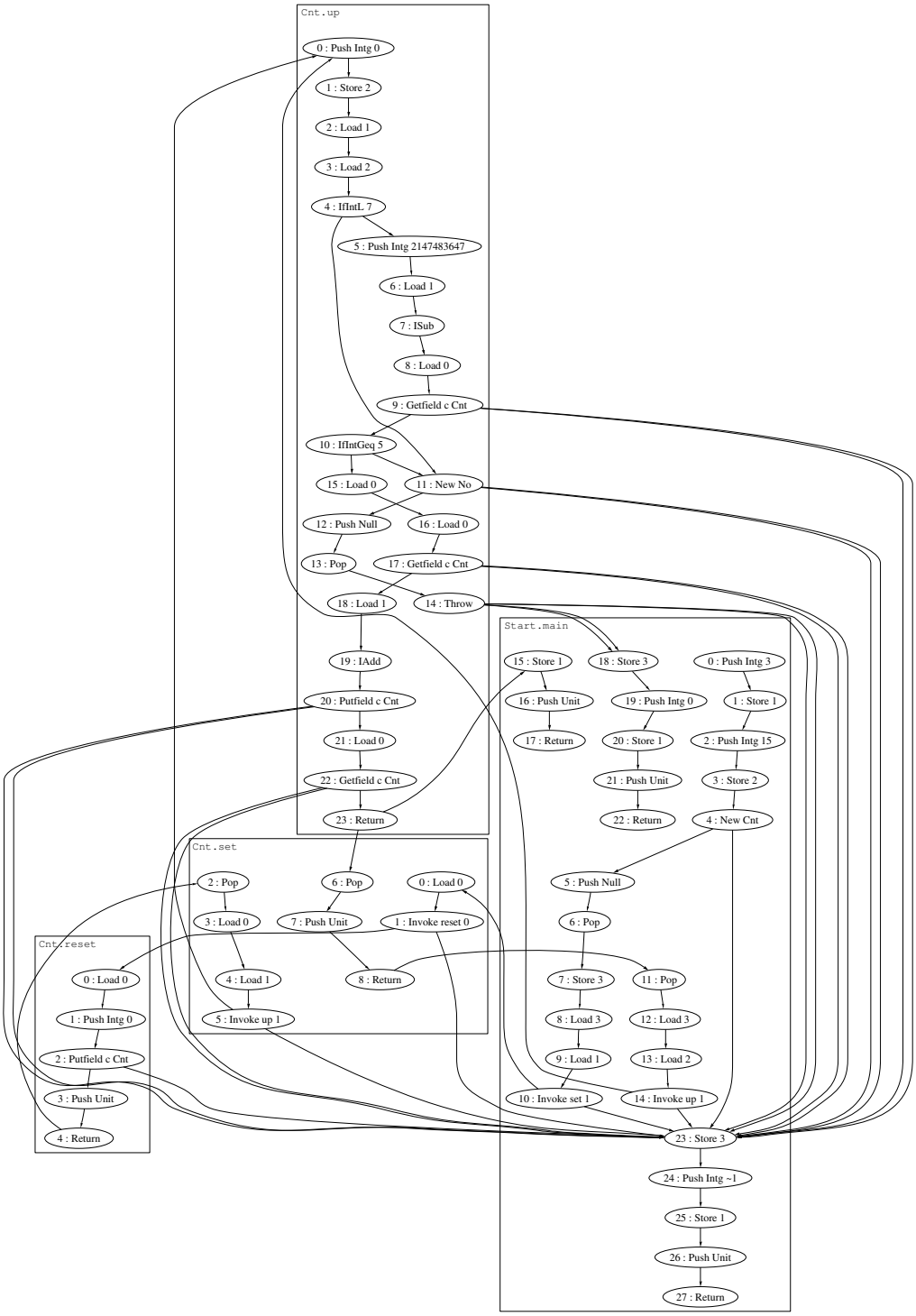


Figure 5.1: Control Flow Graph

As defined in §3.2 positions are triples (C,M,pc) of a class name C, method name M and program counter pc. The code domain, e.g. $domC\ \Pi$, is a list of all positions in a program Π. To collect all these positions we iterate $domCC$ over all classes and $domMC$ over all methods therein. For all classes Cl in Π the code domain $domC\ \Pi$ concatenates the code domains $domCC\ Cl$, which again are concatenations of code domains $domMC\ Mt$ for all methods in Cl.

$domC :: jbc\text{-}prog \Rightarrow pos\ list$

$domC\ \Pi = concat\ (map\ domCC\ (fst\ \Pi))$

$domCC :: jvm\text{-}method\ cdecl \Rightarrow pos\ list$

$domCC\ (C,\ C',\ fs,\ []) = []$
$domCC\ (C,\ C',\ fs,\ (M,\ Ts,\ T,\ m)\ \cdot\ ms) = domMC\ (C,\ M,\ m)\ @$
$\qquad domCC\ (C,\ C',\ fs,\ ms)$

$domMC :: (cname \times mname \times jvm\text{-}method) \Rightarrow pos\ list$
$domMC\ (C,\ M,\ mxs,\ mxl,\ bd,\ et) = map\ (\lambda n.\ (C,\ M,\ n))\ [0..<|bd|]$

Each position p in a program points to a Jinja bytecode instruction. We write $cmd\ \Pi\ p$ to fetch this instruction. If p lies outside the program cmd yields $None$.

$cmd::jbc\text{-}prog \Rightarrow pos \Rightarrow instr\ option$
$cmd\ (P,\ An)\ (C,\ M,\ pc) =$

$(\mathsf{case}\ map\text{-}of\ P\ C\ \mathsf{of}\ None \Rightarrow None$

$\mid \lfloor c \rfloor \Rightarrow \mathsf{case}\ map\text{-}of\ (snd\ (snd\ c))\ M\ \mathsf{of}\ None \Rightarrow None$

$\qquad \mid \lfloor m \rfloor \Rightarrow \mathsf{let}\ (\_,\ \_,\ (\_,\ \_,\ (is,\ \_))) = m\ \mathsf{in}\ \mathsf{if}\ pc < |is|\ \mathsf{then}\ \lfloor is_{[pc]} \rfloor\ \mathsf{else}\ None)$

In wellformed programs - a notion we formalise later on - all classes and all methods of a class have distinct names. Under these circumstances positions in the code domain are exactly those with instructions.

Lemma 5.1 $wf\ \Pi \longrightarrow set\ (domC\ \Pi) = \{p.\ cmd\ \Pi\ p \neq None\}$

Some positions have annotations. In §3.1 we introduced annotated Jinja programs, i.e. $jbc\text{-}prog$, as tuples of bytecode programs as defined in [55] and annotations in form of finite maps from positions to expressions. To fetch the annotation a program Π provides for a position p, we write $anF\ \Pi\ p$. The result is $\lfloor A \rfloor$ or $None$ depending on whether an annotation A exists.

$anF :: jbc\text{-}prog \Rightarrow pos \Rightarrow expr\ option$
$anF\ (P,\ An)\ p = map\text{-}of\ An\ p$

With $domA$ we project the annotated positions out of the code domain.

domA :: *jbc-prog* ⇒ *pos list*
domA Π = [*p*∈*domC* Π . *anF* Π *p* ≠ *None*]

An edge from position p_1 to p_2 with label B indicates a transition from some state (p_1,...) satisfying B to another state (p_2,...) in the operational semantics *effS*. We formalise edges via the successor function *succsF*. Given a program Π and a position p we write *succsF* Π p to obtain the list of all direct successors paired with branch conditions. Branch conditions are expressions in our assertion logic §4 that describe the situations when a particular successor is accessible. In the definition of *succsF* we use separate functions for normal and exceptional successors.

succsF :: *jbc-prog* ⇒ *pos* ⇒ (*pos* × *expr*) *list*

succsF Π *p* = (*case cmd* Π *p of None* ⇒ []
| ⌊*c*⌋ ⇒ *addPos p* (*succsNrm* (Π, *p*, *c*) @ *succsExpt* (Π, *p*, *c*)))

The auxiliary function *addPos* augments the branch conditions with a position formula *Pos p*. This allows to identify subformulas in the verification condition with the part of the program they came frome and helps to locate bugs in case of unprovable verification conditions. In addition *Pos p* ensures a wellformed frames tack, which is important to prove the progress requirement 2.11.

addPos :: *pos* ⇒ (*pos* × *expr*) *list* ⇒ (*pos* × *expr*) *list*
addPos p ps = *map* (λ(*p'*, *B*). (*p'*, *Pos p* ⩓ *B*)) *ps*

The function *succsNrm* yields the successors for normal execution. For `Throw` there are no such successors.

succsNrm (Π, (*C*, *M*, *pc*), `Throw`) = []

Many instructions only have the follow up position (*C*,*M*,*pc+1*) as successor. We write *incP* to increment positions.

incP :: *pos* ⇒ *pos*
incP (*C*, *M*, *pc*) = (*C*, *M*, *pc* + *1*)

The branch condition ⌊*T*⌋ indicates that these successors are always reachable.

c ∈ {`Load` *n*, `Store` *n*, `Push` *v*, `Pop`, `IBin` *bo*, `CmpEq`} ⟶
succsNrm (Π, *p*, *c*) = [(*incP p*, ⌊*T*⌋)]

Jump instructions have relative targets in form of offset numbers t. This avoids labels and uniqueness checks for these. In case of conditional jumps, we get multiple successors with branch conditions expressing the condition or its negation.

succsNrm (Π, (*C*, *M*, *pc*), `Goto` *t*) = [((*C*, *M*, *nat* (*pc* + *t*)), ⌊*T*⌋)]

$succsNrm\ (\Pi,\ (C,\ M,\ pc),\ \texttt{IfIntCmp}\ ro\ t) =$
$[((C,\ M,\ nat\ (pc\ +\ t)),\ Rel\ (St\ 1)\ ro\ (St\ 0)),$
$((C,\ M,\ pc\ +\ 1),\ \lfloor\rfloor\ (Rel\ (St\ 1)\ ro\ (St\ 0)))]$
$succsNrm\ (\Pi,\ (C,\ M,\ pc),\ \texttt{IfFalse}\ t) =$
$[((C,\ M,\ nat\ (pc\ +\ t)),\ St\ 0\ \lfloor=\rfloor\ \lfloor F\rfloor),\ ((C,\ M,\ pc\ +\ 1),\ \lfloor\rfloor\ (St\ 0\ \lfloor=\rfloor\ \lfloor F\rfloor))]$

For instructions that might throw exceptions, *succsNrm* produces a branch condition that excludes this exception. For this purpose we use the auxiliary function *xcpt-cond*, which generates a condition that ensures a particular exception. We define it later, when we come to exceptions.

$succsNrm\ (\Pi,\ p,\ \texttt{Getfield}\ F\ C) = [(incP\ p,\ \lfloor\rfloor\ (xcpt\text{-}cond\ \Pi\ NullPointer\ p))]$
$succsNrm\ (\Pi,\ p,\ \texttt{Putfield}\ F\ C) = [(incP\ p,\ \lfloor\rfloor\ (xcpt\text{-}cond\ \Pi\ NullPointer\ p))]$
$succsNrm\ (\Pi,\ p,\ \texttt{Checkcast}\ C) = [(incP\ p,\ \lfloor\rfloor\ (xcpt\text{-}cond\ \Pi\ ClassCast\ p))]$
$succsNrm\ (\Pi,\ p,\ \texttt{New}\ Cl) = [(incP\ p,\ \lfloor\rfloor\ (xcpt\text{-}cond\ \Pi\ OutOfMemory\ p))]$

Method calls are more complicated, because overriding opens multiple possibilities. It is hard to statically determine the type of the object whose method we are calling. However, we can ask the programmer or compiler to insert annotations constraining the possible types. In §7.1.1 we show how such type annotations can be obtained automatically from the bytecode verifier. For now, assume that every `Invoke` instruction is annotated with such information. Since we have deeply embedded formulas we can analyse annotations syntactically and extract type information from them. This is what the function $extractTy\ (A,ex)$ does.

$extractTy :: expr \times expr \Rightarrow ty\ list$

It searches A for occurrences of subexpressions of the form $Ty\ ex\ tp$ and lists the types tp. It only descends through conjunctions and disjunctions. Although our safety logic does not provide disjunctions, we can introduce these as negated conjunctions.

$\bigvee :: expr\ list \Rightarrow expr$

$\bigvee\ exs = \lfloor\rfloor\ (\bigwedge\ (map\ \lfloor\rfloor\ exs))$

If an annotation enforces a particular type tp in other ways, for example via $\lfloor T\rfloor \Rightarrow Ty\ ex\ tp$, this type will not be extracted. Since we can generate type annotations automatically, expecting them in a particular format, is an acceptable restriction. Before we come to the definition, let us look at the following lemma, which is all we need to know about *extractTy* in our proofs.

Lemma 5.2 When $extractTy(A,ex)$ is not empty, it contains the type of ex under states

where A holds.

$\Pi,s \models\ A \wedge extractTy\ (A,\ ex) = tys \wedge tys \neq [] \longrightarrow (\exists\, tp{\in}set\ tys.\ \Pi,s \models\ Ty\ ex\ tp)$

The formal definition of *extractTy* is a bit technical due to the fact that disjunctions are expressed as negated conjunctions. In contrast to $\bigwedge$ the symbol $\bigvee$ is not a constructor and we cannot use it as a pattern. Instead, we use recursive definitions and directly work on negated conjunctions. First, we introduce *isNegTy* $(ex,\ ex')$, which checks whether ex has the form $\neg\ (Ty\ ex'\ tp)$ for some type tp.

$isNegTy{::}(expr \times expr) \Rightarrow bool$
$isNegTy\ (\neg\ (Ty\ ex\ tp),ex') = (ex = ex')$
$isNegTy\ (\_,ex') = False$

Now, we are ready to define *extractTy* (A,ex). It descends through conjunctions, applies itself on all conjuncts and returns a concatenation of the resulting type lists. Note that if A has conjuncts enforcing incompatible types, then A cannot hold for any state and the result of *extractTy* does not matter. If *extractTy* hits a conjunct of the form $Ty\ ex'\ tp$, it checks whether ex' matches the target expression ex. In case of a match the type tp goes into the result list, otherwise it is ignored. When *extractTy* reaches a conjunct of the form $\neg\ (\neg\ ex' \bigwedge exs)$, it checks whether we have a disjunction of type expressions at hand. This is the case if ex' is a type expression and all expressions in exs are negated type expressions. We collect all possible types by recursively applying *extractTy* on ex' and the remaining disjunction. In all other cases *extractTy* terminates with an empty list.

$extractTy\ (\bigwedge\ [],ex) = []$
$extractTy\ (\bigwedge\ (ex'\#exs),ex) = extractTy\ (ex',ex)\ @\ extractTy\ (\bigwedge\ exs,ex)$
$extractTy\ (Ty\ ex'\ tp,ex) = (if\ ex'{=}ex\ then\ [tp]\ else\ [])$
$extractTy\ (\neg\ (\bigwedge\ []),ex) = []$
$extractTy\ (\neg\ (\bigwedge\ (\neg\ ex'\#exs)),ex) =$
$(if\ (list\text{-}all\ (\lambda\ ex''.\ isNegTy\ (ex'',ex))\ (\neg\ ex'\#exs))$
$\ then\ (extractTy\ (ex',ex)\ @\ extractTy\ (\neg\ (\bigwedge\ exs),ex))\ else\ [])$
$extractTy\ (\_,\ ex) = []$

Having a list of potential types available, we can select the corresponding method entry positions and construct sharp branch conditions. For this purpose we use the auxiliary function *succsInvoke*.

$succsNrm\ (\Pi,\ p,\ \texttt{Invoke}\ Mn\ n) = succsInvoke\ (\Pi,\ Mn,\ n,\ p)$

Using the type extractor function above *succsInvoke* analyses the annotation at p to find out the types of the object reference on top of the stack. It expects this information to be given in form of a disjunction of $Ty\ ex\ tp$ expressions. Then, it constructs exclusive branch conditions for each type. These conditions are exclusive, as $Ty\ ex\ tp$ does not

check subtyping. This is important, because in some situations we may want to exclude some potential successor by not listing the corresponding type in the annotation. For example, if a programmer knows the exact type *tp* of an object *o*, she may annotate a method invocation $o.m()$ with *Ty o tp*. In this case *sucssNrm* only yields one successor and subclasses of *tp* with overridden versions of m do not clutter the verification condition. For each possible type X, it creates a successor position of the form $(C_M,M,0)$, where C_M is for X the closest class in the hierarchy that declares a method with name M. It also constructs a branch condition stating that we actually have a reference of type X, which is not a null pointer.

```
succsInvoke ((P, An), M, n, p) =
(case anF (P, An) p of None ⇒ []
 | ⌊A⌋ ⇒
    concat
     (map (λtp. case tp of
                  Class X ⇒
                    let (C_M, _) = method P X M
                    in [((C_M, M, 0),
                         ⌞⌟ (xcpt-cond (P, An) NullPointer p) ⌞⌟
                         Ty (St n) (Class X))]
                | - ⇒ [])
       (extractTy (A, St n))))
```

For `Return` instructions we scan the code for all positions from which the current method could have been called. The name and class of the current method can be obtained from the position, say (C,M,pc), of the `Return` instruction. Then we scan the code for all positions p' with `Invoke` M n. For each of these call positions p' we construct a branch condition with the annotation and position information of p'. Note that both are put into a *Call* operator in order to evaluate them under a state that satisfied them. As we explain in §6.5 this will lead to modular method verifications.

```
succsNrm (Π, p, Return) =
map (λp'. (incP p', Call (⌞⋀⌟ [aF Π p', Pos p'])) (callers Π p)
```

When instructions throw exceptions control flows to an appropriate handler. The function *succsExpt* knows which exceptions each instruction may throw and invokes *succsXpt* to find potential handlers.

```
succsExpt (Π, p, New C) = succsXpt (Π, OutOfMemory, [p])
succsExpt (Π, p, Getfield F C) = succsXpt (Π, NullPointer, [p])
succsExpt (Π, p, Putfield F C) = succsXpt (Π, NullPointer, [p])
```

$succsExpt\ (\Pi,\ p,\ \texttt{Checkcast}\ C) = succsXpt\ (\Pi,\ ClassCast,\ [p])$

$succsExpt\ (\Pi,\ p,\ \texttt{Invoke}\ M\ n) = succsXpt\ (\Pi,\ NullPointer,\ [p])$

$succsExpt\ (\Pi,\ p,\ \texttt{Throw}) =$
$succsXpt\ (\Pi,\ NullPointer,\ [p])\ @$
$(\mathit{case}\ anF\ \Pi\ p\ \mathit{of}\ None \Rightarrow []$
$\mid \lfloor A \rfloor \Rightarrow concat$
$\quad (map\ (\lambda x.\ \mathit{case}\ x\ \mathit{of}\ Class\ X \Rightarrow succsXpt\ (\Pi,\ X,\ [p])\ \mid\ \text{-} \Rightarrow [])$
$\quad\ (extractTy\ (A,\ St\ 0))))$

$c \in \{\texttt{Load}\ n, \texttt{Store}\ n, \texttt{Push}\ v, \texttt{Return}, \texttt{Pop}, \texttt{IBin}\ no, \texttt{Goto}\ t, \texttt{CmpEq},$
$\quad \texttt{IfIntCmp}\ ro\ t, \texttt{IfFalse}\ t\} \longrightarrow$
$succsExpt\ (\Pi,\ p,\ c) = []$

Handlers are searched by recursively climbing up the call tree and inspecting the exception tables of each call method. The call tree of a position p contains all positions from which one can reach p via method invocations. We discover it with *succsXpt* by recursively tracing *Invoke* instructions backwards. For example Fig. 5.2 shows the call tree of position (*Cnt*,*up*,*0*) from the program in Fig. 5.1.

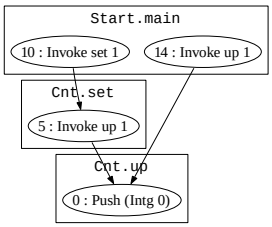


Figure 5.2: Call tree of position (*Cnt*,*up*,*0*)

In *succsXpt* we keep a list of visited positions. When this list becomes too long or empty, *succsXpt* terminates by making all program positions to successors. This means programs with uncaught exceptions usually yield unprovable verification conditions. However, adding a global handler to the main method always helps to avoid this. When *succsXpt* finds a handler it constructs an individual branch condition for this handler. When an exception is caught in the same method as it is thrown (pss=[]), the branch condition is $\lfloor T \rfloor$. Otherwise we restore the call context using *Catch* on the annotation and safety formula of the call point.

$succsXpt\ ((P,\ An),\ X,\ ps) =$
$(\mathit{case}\ |domC\ (P,\ An)| \leq |ps|\ \mathit{of}\ True \Rightarrow map\ (\lambda p.\ (p,\ \lfloor T \rfloor))\ (domC\ (P,\ An))$
$\mid False \Rightarrow$

```
case ps of [] ⇒ map (λp. (p, ⌊⊤⌋)) (domC (P, An))
| p · pss ⇒
    let (C, M, pc) = p; (_, _, (_, _, (_, _, et))) = method P C M;
        A = aF (P, An) p
    in case match-ex-table-e P X pc et of
       None ⇒
         concat
          (map (λp'. succsXpt ((P, An), X, p' · ps))
            (callers (P, An) p))
       | ⌊e⌋ ⇒
          let (f, t, X', pc', d) = e
          in [((C, M, pc'),
               ⨆ ((if pss = [] then [] else [Catch X' A, Catch X (Pos p)]) @
                  [xcpt-cond (P, An) X (last ps)]))]
```

Finally, we define *xcpt-cond*, which yields an expression that describes the reason for a particular exception. All instructions, except `Throw`, only raise one kind of exception. In case of a null reference `Throw` gives a *NullPointer* exception, otherwise the exception reference from the stack. The `New` instruction fails if there is insufficient memory. This is exactly when *NewA 0* evaluates to *Null*. The `Checkcast` C instruction throws a *Classcast* exception if the type of the reference is not a subclass of C. In case of a null reference `Putfield`, `Getfield` or `Invoke` throw *NullPointer* exceptions.

```
xcpt-cond Π X p =
(case cmd Π p of None ⇒ ⌊⊤⌋
 | ⌊c⌋ ⇒
    case c of New C ⇒ NewA 0 ⌈⌈Null⌋⌉
    | Getfield F C ⇒ St 0 ⌈⌈Null⌋⌉
    | Putfield F C ⇒ St 1 ⌈⌈Null⌋⌉
    | Checkcast C ⇒
       ⌈⌉ (St 0 ⌈⌈Null⌋⌉) ⨅
       map
        (λCl. ⌈⌉ (Ty (St 0) (Class Cl)))
        [Cl∈classnames (fst Π) .
         fst Π ⊢ Cl ≼* C]
    | Invoke M n ⇒ St n ⌈⌈Null⌋⌉
    | Throw ⇒ if X = NullPointer
                then (St 0 ⌈⌈Null⌋⌉) ⨆
                     Ty (St 0) (Class X)
                else Ty (St 0) (Class X)
    | - ⇒ ⌊⊤⌋)
```

5.2 Abstract Semantics

In §3.2 we defined the concrete semantics for Jinja bytecode. It describes the behaviour of programs in terms of states and transitions between these. For the purpose of verifying programs we now introduce an abstract semantics. It describes states symbolically with formulas and thus makes them amenable for logical reasoning. To abstract the initial states we introduce *initF* Π, an expression covering all states in *initS*. To simulate transitions, we introduce a weakest precondition operator operator *wpF* Π. This will be the essential engine for our VCG later on.

5.2.1 Initial States

The Jinja virtual machine starts a program Π with one frame on the stack. This frames remains there until the program terminates either with a return from the main method or due to an uncaught exception. Execution starts at position *ipc* Π, which points to the first instruction of the main method. In register 0, which usually contains the `this` reference, we find *Null*. This is because the main method is static. The registers 1 to *mxl*, that is the local variables of the main method, are initialised to some unknown values. By demanding that their evaluation does not yield *None*, we express that the list of registers is long enough. To express this in our logic we introduce *none* as an abbreviation for an ill typed expression, which evaluates to *None* under all states.

$$\begin{aligned}&\mathit{none} :: \mathit{expr}\\&\mathit{none} = ⌞\mathit{Intg}\ \mathit{0}⌟ \sqcup ⌞\mathit{T}⌟\end{aligned}$$

In contrast to the real JVM [88] the Jinja VM [55] preallocates objects for system exceptions. This simplifies the semantics and enables throwing exceptions in situations with no memory left. With *initF* we formalise these constraints as an expression. We use the auxiliary function *addr-of-sys-cxpt* (see [55]) to determine the address a system exception becomes preallocated to.

$$\begin{aligned}&\mathit{sys\text{-}xcptns} :: \mathit{cname\ list}\\&\mathit{sys\text{-}xcptns} = [\mathit{NullPointer}, \mathit{ClassCast}, \mathit{OutOfMemory}]\\&\mathit{initF} :: \mathit{jbc\text{-}prog} \Rightarrow \mathit{expr}\\&\mathit{initF}\ \Pi =\\&⌞\bigwedge⌟\ ([\mathit{Pos}\ (\mathit{ipc}\ \Pi), \mathit{Rg}\ \mathit{0} \sqsupseteq ⌞\mathit{Null}⌟, \mathit{FrNr} \sqsupseteq ⌞\mathit{Intg}\ \mathit{1}⌟] @\\&\quad \mathit{map}\ (\lambda C.\ \mathit{Ty}\ ⌞\mathit{Addr}\ (\mathit{addr\text{-}of\text{-}sys\text{-}xcpt}\ C)⌟\ (\mathit{Class}\ C))\ \mathit{sys\text{-}xcptns} @\\&\quad (\mathsf{let}\ (C, M, \mathit{pc}) = \mathit{ipc}\ \Pi; (\_, \_, (\_, \_, (\mathit{mxl}, \_))) = \mathit{method}\ (\mathit{fst}\ \Pi)\ C\ M\\&\quad\ \mathsf{in}\ \mathit{map}\ (\lambda n.\ ⌐\ (\mathit{Rg}\ n \sqsupseteq \mathit{none}))\ [\mathit{1}..\mathit{mxl}]))\end{aligned}$$

5.2.2 Transitions

The purpose of the *wpF* operator is to simulate transitions on states with transformations of expressions. Our aim is a function that preserves the evaluation of expressions Q by compensating effects of the concrete semantics with substitutions on Q. Formally, this property can be described as follows:

$\ldots \wedge ((p,\ m,\ e),\ (p',\ m',e')) \in \mathit{effS}\ \Pi \longrightarrow$
$(\mathit{evalE}\ \Pi\ (p,m,e)\ (\mathit{wpF}\ \Pi\ p\ p'\ Q) = \mathit{evalE}\ \Pi\ (p',\ m',e')\ Q)$

This relation between abstract and concrete semantics only holds with some restrictions, which we suppress with $\ldots$. For now, just assume that that $\mathit{wpF}\ \Pi\ p\ p'\ Q$ needs to transform a postcondition Q, such that it evaluates to the same value as Q does in the successor state. It substitutes all subexpressions of Q whose evaluation would change due to the state transition by another expression. The new expression must evaluate under the current state to the same value as the replaced subexpression does under the successor state. For this purpose we use a substitution function

$\mathit{substE}::(\mathit{expr} \sim\sim> \mathit{expr}) \Rightarrow \mathit{expr} \Rightarrow \mathit{expr}$,

which maintains an expression map *em*. We will define *substE em ex* such that it traverses a given expression *ex* (not descending into temporal operators) and simultaneously replaces all instances of expressions that appear on the left hand side of a maplet in *em* by the corresponding right hand side. Example:

$\mathit{substE}\ [(\mathit{St}\ 0,\mathit{Rg}\ 0)]\ (\mathit{St}\ 0 \sqsubset\sqsupset \mathit{Call}\ (\mathit{St}\ 0)) = \mathit{Rg}\ 0 \sqsubset\sqsupset \mathit{Call}\ (\mathit{St}\ 0)$

The substitution does not only replaces variables, because Jinja Bytecode instructions may also change the heap and other parts of the state. Hence, we sometimes have to substitute entire expressions. In the definition of *wpF* we analyse the postcondition and extract subexpressions of particular patterns. These are then used to build maplets for the substitution map. For example, we start our definition of *wpF* by constructing a map *pm* adjusting position expressions.

$\mathit{wpF} :: \mathit{jbc\text{-}prog} \Rightarrow \mathit{pos} \Rightarrow \mathit{pos} \Rightarrow \mathit{expr} \Rightarrow \mathit{expr}$
$\mathit{wpF}\ \Pi\ p\ p'\ Q =$
$(\mathsf{let}\ \mathit{pm} = \mathit{map}\ (\lambda q.\ (\mathit{Pos}\ q,\ \mathsf{if}\ q = p'\ \mathsf{then}\ \mathit{Pos}\ p\ \mathsf{else}\ \lfloor F \rfloor))\ (\mathit{getPosEx}\ Q)$
$\ \mathsf{in\ case}\ \mathit{cmd}\ \Pi\ p\ \mathsf{of}\ \mathit{None} \Rightarrow \lfloor F \rfloor$
$\quad \mid \lfloor i \rfloor \Rightarrow \mathsf{case}\ \mathit{handlesEx}\ (\mathit{fst}\ \Pi)\ p'\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{wpNrm}\ \Pi\ p\ p'\ Q\ \mathit{pm}\ i$
$\qquad\quad \mid \lfloor cn \rfloor \Rightarrow \mathit{wpExc}\ \Pi\ p\ p'\ Q\ \mathit{pm}\ \mathit{cn}\ i)$

In $\mathit{wpF}\ \Pi\ p\ p'\ Q$ we have to compensate the change of the program counter from p to p'. This affects all position expressions $\mathit{Pos}\ q$ in Q, which we extract with *getPosEx Q*. If q differs from p', the position formula $\mathit{Pos}\ q$ is false in the successor state and must be so in the current state. We replace it with $\lfloor F \rfloor$. Otherwise, if $q = p'$ the position formula

Pos q is true if all positions on the frame stack form a call chain. Note that this is also checked with *Pos* and holds in the successor state exactly when *Pos p* holds in the predecessor state. The reason for having the positions p and p' among the arguments of *wpF* is that instructions may behave differently on each outgoing control flow edge. In case of Jinja, we have instructions that can throw exceptions. To cope with that we use two different weakest precondition operators *wpNrm* and *wpExc*. We can determine statically from p' which one to apply, thus *wpF* does not have to introduce disjunctions of both effects. If p' is the entry position of some handler for an exception X, we assume that the transition from p to p' is due to an exception and delegate work to *wpExc*. Otherwise we use *wpNrm*, which transforms Q according to normal execution. In our wellformedness constraint *wf* Π we demand that all exception handlers have distinct entry positions, a condition that holds for all bytecode programs compiled from Java sources. Due to the unique entry positions, we can define a function *handlesEx P p'*, which yields $\lfloor X \rfloor$ if p' starts the handler catching exceptions of type *Class X* or *None* if no such handler exists. We skip the formal definition of *handlesEx* as it is a straightforward lookup in all exception handler tables on the call tree. However, before we turn our attention on the actual transformations of *wpNrm* and *wpExc*, we go into the details of substitution and extractor functions.

Substitution

In many textbooks on Hoare Logics [99, 72] substitution is not a big deal as only program variables need to be substituted. In case of Jinja a state has many facets and much more different kinds of expressions need to be substituted. In particular the heap complicates matters as it demands to substitute even composed expressions. Instead of having multiple substitution functions each handling the effect on a particular part of the state, we decided to define a uniform operation. Since we do not distinguish between expressions and formulas, one map from expressions to expressions does the job. The substitution *substE em ex*, which we define formally in Fig. 5.3 transforms the expression *ex* according to the maplets in *em*. For every kind of expression *substE em ex* first checks if *em* provides a maplet for *ex*. If so, it becomes substituted by its mapping and the substitution stops. Otherwise, *substE* recursively descends into all subexpressions, except for those below temporal operators, i.e. *Call* and *Catch*. This is because most instructions only modify the topmost frame, which has no influence on the evaluation of temporal expressions.

```
substE:: (expr ~~> expr) ⇒ expr ⇒ expr
substE em ex =
(case ex of ⌊⋀⌋ es ⇒ ⌊⋀⌋ (map (substE em) es)
 | - ⇒ case map-of em ex of
        None ⇒
          case ex of Gf F C e ⇒ Gf F C (substE em e)
          | Num e1 no e2 ⇒ Num (substE em e1) no (substE em e2)
          | Rel e1 ro e2 ⇒ Rel (substE em e1) ro (substE em e2)
          | ⌊if⌋ b ⌊then⌋ t ⌊else⌋ e ⇒ ⌊if⌋ substE em b ⌊then⌋ substE em t ⌊else⌋ substE em e
          | e1 ⌊=⌋ e2 ⇒ substE em e1 ⌊=⌋ substE em e2 | ⌊¬⌋ e ⇒ ⌊¬⌋ (substE em e)
          | e1 ⌊⇒⌋ e2 ⇒ substE em e1 ⌊⇒⌋ substE em e2 | ⌊∀⌋ v e ⇒ ⌊∀⌋ v (substE em e)
          | Ty e tp ⇒ Ty (substE em e) tp | - ⇒ ex
        | ⌊ex'⌋ ⇒ ex')
```

Figure 5.3: Substitution of expressions

Extractions

Since we defined substitutions via finite maps, we can combine them using list concatenation and perform induction on lists. The expression maps also allow us to substitute composed expressions at once. On the other hand, finite maps also come with a price. We cannot represent substitutions with infinitely many patterns. For example shifting the stack by replacing all occurrences of *St k* with *St (k+1)* is something we cannot directly represent. However, we just need *substE mp ex* to construct weakest preconditions for postconditions *Q*. For that *wpF* only has to substitute some subexpressions in *Q*. These are finitely many and we can extract those from *Q* and turn them into a finite map of substitution maplets. Since different instructions affect different kinds of subexpressions, we need various extractions. Many of these work in a similar way. They traverse an expression and collect all subexpressions fitting a given pattern. To avoid defining this traversal separately for each extractor function Fig. 5.4 introduces a general fold operator *foldE f c a ex*. In addition to the analysed expression *ex* it takes three parameters, *f*, *c* and *a*. With *a* one hands over a default result for atomic expressions *ex*. The function *f* performs the actual work on expressions. It gets the current subexpression and the current result as arguments and produces a new result. The combinator function *c* is used to combine results from different recursive calls of *foldE*. For example if *ex* is an expression with multiple subexpressions *foldE* applies itself on all subexpressions and combines the results to one final result using *c*.

Using *foldE* we can define the function *getPosEx ex*, which extracts all position expressions inside *ex*, except those below a *Catch* or *Call* operator. For example:

getPosEx (*And* [*Pos p*,⌊*T*⌋ ⌊⇒⌋ *Pos q*,*Call* (*Pos c*)]) = [*p*,*q*]

```
foldE:: (expr × 'a ⇒ 'a) ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ 'a ⇒ expr ⇒ 'a
foldEs:: (expr × 'a ⇒ 'a) ⇒ ('a ⇒ 'a ⇒ 'a) ⇒ 'a ⇒ expr list ⇒ 'a

foldEs f c a [] = a
foldEs f c a (e · es) = c (foldE f c a e) (foldEs f c a es)
foldE f c a ex =
(case ex of Gf F C e ⇒ f (ex, foldE f c a e)
 | Num e1 no e2 ⇒ f (ex, c (foldE f c a e1) (foldE f c a e2))
 | Rel e1 ro e2 ⇒ f (ex, c (foldE f c a e1) (foldE f c a e2))
 | ⌊if⌋ b ⌊then⌋ t ⌊else⌋ e ⇒ f (ex, c (c (foldE f c a b) (foldE f c a t)) (foldE f c a e))
 | e1 ⌊=⌋ e2 ⇒ f (ex, c (foldE f c a e1) (foldE f c a e2))
 | ⌊¬⌋ e ⇒ f (ex, foldE f c a e)
 | e1 ⌊⇒⌋ e2 ⇒ f (ex, c (foldE f c a e1) (foldE f c a e2))
 | ⌊⋀⌋ es ⇒ f (ex, foldEs f c a es) | ⌊∀⌋ v e ⇒ f (ex, foldE f c a e)
 | Ty e tp ⇒ f (ex, foldE f c a e) | Call e ⇒ f (ex, foldE f c a e)
 | Catch X e ⇒ f (ex, foldE f c a e) | - ⇒ f (ex, a))
```

Figure 5.4: Fold operator for expressions

We use the extracted expressions to define a substitution that handles the effect of an instruction on the state. Since most instructions only modify the topmost frame, we usually do not have to modify subexpressions inside a *Catch X ex* or *Call ex* expression. All reachable states s, i.e. $s \in$ *Reachables* Π, have non-empty frame stacks and *Catch* or *Call* remove at least one frame, before their argument expression becomes evaluated. To avoid extractions below *Catch* and *Call* we introduce the filter function *noCC ex exs*, which keeps all results in *exs* when *ex* is neither *Catch* nor *Call*, or removes them otherwise.

```
noCC:: expr × 'a list ⇒ 'a list
noCC (ex, as) = (case ex of Call ex' ⇒ [] | Catch C ex' ⇒ [] | - ⇒ as)
```

The auxiliary function *posEx ex* is our actual extractor function. It checks, whether *ex* is of the form *Pos p*. If so, it yields $[p]$, otherwise [].

```
posEx:: expr ⇒ pos list
posEx ex = (case ex of Pos p ⇒ [p] | - ⇒ [])
```

Using *foldE* we now drive *posEx* inside a given expression *ex*. As combinator c we use list concatenation @, which collects all results found for any subexpression of *ex*. Whenever *foldE* reaches a subexpression *se* it applies the function λ (*se*,*as*). *posEx es* @ (*noCC* (*ex*,*as*)). This function receives the current subexpression *se* and a list of all results obtained from children of *se*. We feed *as* into *noCC se as* and in order to drop all these results when *se* is *Catch* or *Call*. To the resulting list we append *posEx se*, which gives us a new position $[p]$ when *ex* is of the form *Pos p*. The list *posEx se* @ *noCC*(*se*,*as*)

contains all extracts found in *se* and we propagate it upwards.

getPosEx:: *expr* ⇒ *pos list*
getPosEx ex = *foldE* (λ(*ex*, *as*). *posEx ex* @ *noCC* (*ex*, *as*)) (@) [] *ex*

For the remaining extractor functions we only give brief explanations. They work analogously to *getPosEx*. Quite often, we have to extract the indices of stack and register expressions. For this purpose, we introduce *stkIds* and *rgIds*.

stkId:: (*expr* × *var list*) ⇒ *var list*
stkId (*ex*, *vs*) = (*case ex of St k* ⇒ [*k*] | - ⇒ *vs*)
stkIds:: *expr* ⇒ *var list*
stkIds ex = *foldE* (λ(*ex*, *as*). *noCC* (*ex*, *stkId* (*ex*, *as*))) (@) [] *ex*

rgId:: (*expr* × *var list*) ⇒ *var list*
rgId (*ex*, *vs*) = (*case ex of Rg k* ⇒ [*k*] | - ⇒ *vs*)
rgIds:: *expr* ⇒ *var list*
rgIds ex = *foldE* (λ(*ex*, *as*). *noCC* (*ex*, *rgId* (*ex*, *as*))) (@) [] *ex*

To extract temporal expressions, we use *getCatchEx* and *getCallEx*.

callEx:: *expr* ⇒ *expr list*
callEx ex = (*case ex of Call ex′* ⇒ [*ex′*] | - ⇒ [])
getCallEx:: *expr* ⇒ *expr list*
getCallEx ex = *foldE* (λ(*ex*, *as*). *callEx ex* @ *noCC* (*ex*, *as*)) (@) [] *ex*

catchEx:: *expr* ⇒ (*cname* × *expr*) *list*
catchEx ex = (*case ex of Catch cn ex′* ⇒ [(*cn*, *ex′*)] | - ⇒ [])
getCatchEx:: *expr* ⇒ (*cname* × *expr*) *list*
getCatchEx ex = *foldE* (λ(*ex*, *as*). *catchEx ex* @ *noCC* (*ex*, *as*)) (@) [] *ex*

Sometimes we need to find expressions depending on the heap. We introduce a separate datatype, i.e. *heapexpr* = *GF vname cname expr* | *TY expr ty* , to subsume expressions that might be affected by a field update.

gfEx:: (*vname* × *cname* × *expr*) ⇒ *expr list*
gfEx (*F*, *C*, *ex*) = (*case ex of Gf F′ C′ ex′* ⇒ *if F* = *F′* ∧ *C* = *C′ then* [*ex′*] *else* [] | - ⇒ [])
getGfEx:: *vname* ⇒ *cname* ⇒ *expr* ⇒ *expr list*
getGfEx F C ex = *foldE* (λ(*ex*, *as*). *as* @ *gfEx* (*F*, *C*, *ex*)) (@) [] *ex*

heapEx :: *expr* ⇒ *heapexpr list*
heapEx ex = (*case ex of Gf F C ex′* ⇒ [*GF F C ex′*] | *Ty ex′ tp* ⇒ [*TY ex′ tp*] | - ⇒ [])
getHeapEx :: *expr* ⇒ *heapexpr list*
getHeapEx ex = *foldE* (λ(*ex*, *as*). *as* @ *heapEx ex*) (@) [] *ex*

Apart from field updates also the creation of new objects modifies the heap. With

getNewEx we extract all indices n of *NewA* n expressions.

newEx :: *expr* ⇒ *nat list*
newEx ex = (*case ex of NewA n* ⇒ [*n*] | - ⇒ [])
getNewEx :: *expr* ⇒ *nat list*
getNewEx ex = *foldE* (λ(*ex*, *as*). *newEx ex* @ *as*) (@) [] *ex*

Equipped with extractor functions, we now describe how the normal behaviour of instructions can be described in our assertion logic. We handle this with the *wpNrm* function.

wpNrm :: *jbc-prog* ⇒ *pos* ⇒ *pos* ⇒ *expr* ⇒ (*expr* ∼∼> *expr*) ⇒ *instr* ⇒ *expr*

Argument Passing

The `Load` n instruction loads the value of register n onto the stack. We replace *St 0* with *Rg n* and decrement the index of all other stack expressions. The latter simulates the growth of the stack. For example the value at the third stack position after the Load equals the one at position 2 before. The effect on the program counter is already handled by *pm*, which we *wpF* creates and passes over to *wpNrm*.

wpNrm Π *p p′ Q pm* (`Load` *n*) = *substE* (*pm*@
(*map* (λ*k*. (*St k*,*if k*=*0 then Rg n else St* (*k* − *1*))) (*stkIds Q*))) *Q*

The `Store` n instruction removes the topmost value from the stack and stores it in register n. We replace *Rg n* with *St 0* and increment all stack positions to compensate the stack shift.

wpNrm Π *p p′ Q pm* (`Store` *n*) =
substE (*pm*@((*Rg n*,*St 0*) · *map* (λ*k*. (*St k*,*St* (*k*+*1*))) (*stkIds Q*))) *Q*

The `Push` v instruction pushes value v onto the stack. We simulate this by replacing *St 0* with ␣ and decrementing all other stack positions.

wpNrm Π *p p′ Q pm* (`Push` *v*) = *substE* (*pm*@
(*map* (λ*k*. (*St k*,*if k*=*0 then* ␣ *else St* (*k* − *1*))) (*stkIds Q*))) *Q*

Arithmetics, Checks and Branches

The `IBin` *no* instruction performs binary operations on integers. We replace *St 0* with the corresponding arithmetic expression and shift the other stack indices to compensate the removal of 2 arguments.

wpNrm Π *p p′ Q pm* (`IBin` *no*) = *substE* (*pm*@(*map* (λ*k*. (*St k*,*if k*=*0*
then Num (*St 1*) *no* (*St 0*) *else* (*St* (*k*+*1*)))) (*stkIds Q*))) *Q*

The `IfIntCmp` *ro t* instruction removes the topmost two integer values from the stack, compares them and branches accordingly. To simulate the removal of the two stack elements, we shift stack expressions.

wpNrm Π *p p′ Q pm* (`IfIntCmp` *ro t*) =
substE (*pm*@(*map* (λ*k*. (*St k*,*St* (*k*+*2*))) (*stkIds Q*))) *Q*

The direct jump `Goto` *t* has no effect except from changing the program counter, which is already handled by *pm*.

wpNrm Π *p p′ Q pm* (`Goto` *t*) = *substE pm Q*

The instruction `CmpEq` removes the topmost two values from the stack, compares them and pushes the result in form of a boolean value back. We replace *St 0* with the condition *St 0* $\llcorner\!\!=\!\!\lrcorner$ *St 1*. The new stack is one element shorter and we compensate this by incrementing all remaining stack expressions.

wpNrm Π *p p′ Q pm* `CmpEq` = *substE* (*pm*@
(*map* (λ*k*. (*St k*,*if k*=*0 then St 0* $\llcorner\!\!=\!\!\lrcorner$ *St 1 else St* (*k*+*1*))) (*stkIds Q*))) *Q*

The `IfFalse` *t* instruction expects a boolean on top of the stack, removes it and branches accordingly. We only have to simulate the shortening of the stack, the position change is handled by *pm*.

wpNrm Π *p p′ Q pm* (`IfFalse` *t*) = *substE* (*pm*@(*map* (λ*k*. (*St k*,*St* (*k*+*1*))) (*stkIds Q*))) *Q*

The `Checkcast` *C* instruction throws an exception if the topmost stack value is not a reference to an object of class *C*. Otherwise it just skips. Since *wpNrm* only handles non-exceptional behaviour, it just needs to change position expressions according to *pm*.

wpNrm Π *p p′ Q pm* (`Checkcast` *Cl*) = *substE pm Q*

In case of `Throw` there is always an exception. Since we cannot reach p' with a "normal" transition, *wpNrm* yields $\llcorner F \lrcorner$.

wpNrm Π *p p′ Q pm* `Throw` = $\llcorner F \lrcorner$

Heap Access

Three Jinja instructions, namely `Getfield`, `Putfield` and `New` depend on or modify the heap. The instruction `Getfield` *F C* expects a reference to an object of class *C* on top of the stack and replaces it with the value of field *F* of this object. To simulate this statically, we only have to replace *St 0* with the fetched field value.

wpNrm Π *p p′ Q pm* (`Getfield` *F C*) = *substE* (*pm*@[(*St 0*,*Gf F C* (*St 0*))]) *Q*

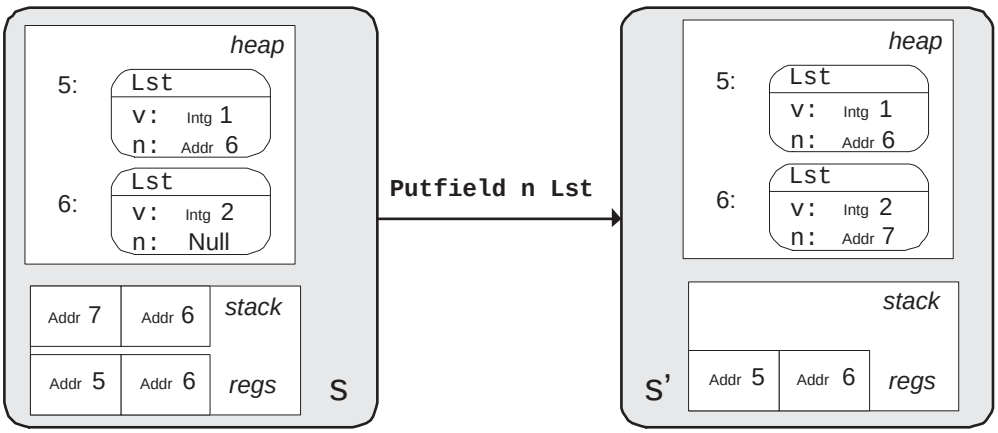


Figure 5.5: Putfield

i	Q_i	$evalE\ \Pi\ s'\ Q_i$
1	$Gf\ n\ Lst\ (\lfloor Addr\ 6 \rfloor)$	$\lfloor Addr\ 7 \rfloor$
2	$Gf\ n\ Lst\ (Rg\ 1)$	$\lfloor Addr\ 7 \rfloor$
3	$Gf\ n\ Lst\ (Rg\ 0)$	$\lfloor Addr\ 6 \rfloor$
4	$Gf\ n\ Lst\ (Gf\ n\ Lst\ (Rg\ 0))$	$\lfloor Addr\ 7 \rfloor$
5	$Gf\ v\ Lst\ (Gf\ n\ Lst\ (Rg\ 0))$	$\lfloor Intg\ 2 \rfloor$
6	$Rg\ 0$	$\lfloor Addr\ 5 \rfloor$
7	$Rg\ 1$	$\lfloor Addr\ 6 \rfloor$

Figure 5.6: Example expressions

The `Putfield` F C instruction expects a reference followed by a value on top of the stack. It removes both and updates field F of the referenced object of class C with this value. The removal of two operands can be simulated by shifting indices of *St* expressions. With *em* we introduce a substitution map that adds this part to *pm*, the map handling position expressions.

$em{=}pm@(map\ (\lambda k.\ (St\ k{,}St\ (k{+}2)))\ (stkIds\ Q))$

The heap modification, although simple in the dynamic semantics, causes trouble for our static simulation. First of all it is difficult, if not impossible, to figure out the address of the modified object. Even if we could sharply approximate this runtime value, aliasing comes into play. Many syntactically different expressions may depend on the modified field.

In Fig. 5.5 we have a `Putfield` that updates field n of an *Lst* object residing at address *6*. The old field value *Null* is changed to *Addr 7*. To illustrate the following transformations Fig. 5.6 shows a few expressions and their evaluation under s'.

Expression Q_1 accesses the modified field using the concrete address of the object. Replacing Q_1 by *St 0* would work in this case, but not in general. The fact that *Addr 6* equals *St 1* is just a coincidence and does not hold for all states. However, we can use conditionals to handle `Putfield`'s effect. Using $\llbracket if\ \_\ then\ \_\ else\ \_ \rrbracket$ we can check whether an expression *Gf F C ex* accesses the modified field and return the new or old value accordingly. This means *wpNrm* transforms every expression Q_i from Fig. 5.6 into W_i:

$W_i = wpNrm\ \Pi\ p\ p'\ Q_i\ pm\ (\texttt{Putfield}\ v\ Lst)$

For Q_1 we get the following result.

$W_1 = \llbracket if\ \lfloor Addr\ 6 \rfloor = St\ 1\ then\ St\ 0\ else\ Gf\ n\ Lst\ (\lfloor Addr\ 6 \rfloor) \rrbracket$

Evaluated under s expression W_1 yields $\lfloor Addr\ 7 \rfloor$, the same as Q_1 does under s'. Analogously, *wpNrm* transforms Q_2 into W_2 and Q_3 into W_3.

$W_2 = \llbracket if\ Rg\ 1 = St\ 1\ then\ St\ 0\ else\ Gf\ n\ Lst\ (Rg\ 1) \rrbracket$

$W_3 = \llbracket if\ Rg\ 0 = St\ 1\ then\ St\ 0\ else\ Gf\ n\ Lst\ (Rg\ 0) \rrbracket$

In Q_4 we find nested field lookups, hence *wpNrm* introduces nested conditionals. We abbreviate the result by reusing W_3.

$W_4 = \llbracket if\ W_3 = St\ 1\ then\ St\ 0\ else\ Gf\ n\ Lst\ W_3 \rrbracket$

In the real result W_3 would be expanded, which causes exponential growth of the expression in the depth of nested field lookups. For each *Gf F C ex* expression the constructed conditional contains the address expression *ex* (in transformed form) twice: once for the check and once in the *else* branch. Should this exponential growth turn out to be a problem in practice one can tackle it by using *Let* expressions. In an extended version

of our assertion language we plan to support primitive recursion, which would allow to express a *Let* operator.
Note that *wpNrm* does not need to protect $Gf\ F'\ C'\ ex$ expressions where F' or C' differs from F or C in the `Putfield` $F\ C$ instruction. For example in Q_5 the access of field v, which differs from n, must not be protected by a conditional.

$W_5 = Gf\ v\ Lst\ W_3$

In that respect our approach is quite similar to the split heap approach [26, 24, 61], where modifications are kept local by keeping fields in separate heaps. If we had not used a deeply embedded assertion language this static distinction of field names in expressions would not be possible.
The formal definition of our transformation turns out to be tricky. For every $Gf\ F\ C\ ex$ expression in the postcondition Q we need the entire substitute in our map. To construct this substitute we first have to transform other $Gf\ F\ C$ expressions inside *ex*. To accomplish this task, we first extract all $Gf\ F\ C$ expressions in Q in sub-term order. That is expressions that occur as subexpressions in other expressions come first. For example here is what *getGfEx* yields for Q_4:

$L = getGfEx\ n\ Lst\ Q_4 = [Rg\ 0, Gf\ n\ Lst\ (Rg\ 0)]$.

The expressions *getGfEx* extracts are so-called *address* expressions, because *field lookup* expressions use them locate objects. Note that in L the atomic address expression $Rg\ 0$ comes first. Next, we need the left oriented fold operator on lists, i.e *foldl*.

$foldl :: ('a \Rightarrow 'b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b\ list \Rightarrow 'a$
$foldl\ f\ a\ [] = a$
$foldl\ f\ a\ (x \cdot xs) = foldl\ f\ (f\ a\ x)\ xs$

Using *foldl* on L we construct *gfe*, a map substituting field lookups in Q_4.

$$\begin{aligned} gfe = foldl\ (\lambda mp\ ex.&\textsf{let}\ ex'{=}substE\ mp\ ex \\ &\textsf{in}\ (Gf\ F\ C\ ex, \textsf{if}\ (ex' = St\ 1) \\ &\qquad \textsf{then}\ St\ 0 \\ &\qquad \textsf{else}\ Gf\ F\ C\ ex') \cdot mp)\ em\ L \\ = [(Gf\ n\ Lst\ (Rg\ 0), W_3), &(Gf\ n\ Lst\ (Gf\ n\ Lst\ (Rg\ 0)), W_4)] \end{aligned}$$

Note that W_3 is needed to compute W_4. This means we first have to construct the substitutes for the smaller address expressions and use those to construct the bigger ones. This works, because of the sub-term order of L. When *foldl* takes the next expression from L all maplets for its subexpressions are already produced and in *mp*. When we apply *gfe* to Q_4 only the mapping for the outermost $Gf\ n\ Lst$ expression matters. When *substE* reaches it, it instantly performs the substitution and does not descend further. Now, we are ready to assemble the full definition of *wpNrm*. We apply *remdup*, a function

removing duplicates from lists, on the list of extracted expressions. This simplifies our proof later on, which performs inductions on maps represented as lists. Only if the left hand sides of all maplets are distinct, removing one element from the list also reduces the domain of the map.

$$
\begin{array}{l}
wpNrm\ \Pi\ p\ p'\ Q\ pm\ (\texttt{Putfield}\ F\ C) = (\mathit{let} \\
em = pm@(map\ (\lambda k.\ (St\ k, St\ (k{+}2)))\ (stkIds\ Q)); \\
gfe = foldl\ (\lambda mp\ ex.\ \mathit{let}\ ex' = substE\ mp\ ex \\
\qquad\qquad \mathit{in}\ (Gf\ F\ C\ ex, \mathit{if}\ (ex' = St\ 1) \\
\qquad\qquad\qquad\qquad \mathit{then}\ St\ 0\ \mathit{else}\ Gf\ F\ C\ ex') \cdot mp) \\
\qquad em\ (remdup\ (getGfEx\ F\ C\ Q)) \\
\mathit{in}\ substE\ gfe\ Q)
\end{array}
$$

Another instruction that modifies the heap is `New` C, which creates an object of class C and pushes the reference to it onto the stack. Here we have the difficulty that the new state contains a value (a new address) that is not present in the old state. Hence, it is difficult to construct an expression yielding this new value when evaluated under the current state. To solve this problem we introduced the *NewA* n expression, which anticipates object allocation by yielding the address the n-th new object would have. For example *NewA 0* yields the next free address in the heap, the one `New` *Cl* would allocate and push onto the stack. By replacing *St 0* with *NewA 0* in a postcondition Q, we preserve the evaluation of subexpressions in Q that talk about this new address. To compensate the growth of the stack, we shift the indices of other *St* k expressions and replace them with *St* $(k - 1)$. The index n in our *NewA* n expression is necessary, because the postcondition Q may already contain other expressions of the form *NewA*. Since newly allocated addresses differ from all other addresses we must distinguish these expressions. In the state after `New` *Cl* we have one more object allocated than before. When we evaluate *NewA* k in this state, we obtain the same address as when we evaluate *NewA* $(k{+}1)$ in the state before. Hence, we replace all occurrences of *NewA* k in Q with *NewA* $(k{+}1)$. Apart from that we have to deal with expressions that could possibly access fields of the newly created object. For example this could be the case for expressions like Q_1 in Fig. 5.6, which fetches the field of an object at a concrete address. Since `New` *Cl* initialises all fields with default values, we can replace such expressions with default constant expressions. For example if we find a *Gf v Lst* (*St 0*) in Q, we can replace it with dv, where dv is the default value for integer fields. To obtain these default values, we first construct a default object via *blank P Cl* (see [55]) and then fetch the corresponding field. Since we can not distinguish address expressions statically, we again employ conditionals for this purpose. The construction with the *foldl* operator works just like in the `Putfield` case. Apart from field fetching expressions also type checking expressions can be affected by the `New` C operation. It can be that in *Ty ex tp* the expression *ex* evaluates to the address of the newly created object. If so, then the validity of the expressions depends on whether *tp* equals *Class C*. Otherwise the

validity of the expression is unaffected. We can handle this situation similarly to the field fetching expressions by introducing conditionals.

```
wpNrm Π p p' Q pm (New Cl) = (let
em = (pm@(map (λk. (St k,if k=0 then NewA 0
                              else St (k − 1))) (stkIds Q))@
      (map (λn. (NewA n, NewA (n+1))) (getNewEx Q)));
gfe' = foldl (λmp hex. (case hex
                        of GF F C ex ⇒ (let ex'=substE mp ex
                                        in (Gf F C ex,⌊if ex' ≠ NewA 0
                                                      ⌊then⌋ ⌊the ((snd (blank (fst Π) Cl))(F,C))⌋
                                                      ⌊else⌋ Gf F C ex'))
                        | TY ex ty   ⇒ (let ex'=substE mp ex
                                        in (Ty ex ty,⌊if ex' ≠ NewA 0
                                                      ⌊then⌋ ⌊Bool ((Class Cl) = ty)⌋
                                                      ⌊else⌋ Ty ex' ty))) · mp)
          em (remdup (getHeapEx Q))
in substE gfe' Q)
```

Method Call and Returns

Method calls and returns modify the frame stack and not just the topmost frame. Here our temporal operators *Call* and *Catch* come into play. With them we can move down the frame stack and restore previous states. The `Invoke` $M\ n$ instruction allocates a new frame, and copies $n+1$ values from the stack of the call frame into registers *0* to n in reversed order. The value lying on stack position *1* moves into register n, that from position *2* into register $n-1$ and so on. Register *0* becomes the *this* pointer, which lies on top of the operand stack at call time.
In Fig. 5.7 we illustrate the effect of the `Invoke` *up 1* instruction from position $(Cnt,set,5)$ of the program shown in Fig. 3.3. Since the frame stack grows, we replace *FrNr* with *FrNr* ⌊+⌋ ⌊*Intg 1*⌋. Registers *Rg 0* to *Rg n* become substituted with the corresponding stack elements as described above. In Fig. 5.7 we have $eval\ \Pi\ s'\ (Rg\ 1) = \lfloor Intg\ 3 \rfloor = eval\ s\ (St\ 0)$. The registers for the local variables, that is $Rg\ (n+1)$ to $Rg\ mxV$, become initialised with *arbitrary*. Registers above mxV are illegal (outside the bank) and we replace them with the *none* expression. In our example mxV of *up* is *2*. Hence, *Rg 3* would evaluate to *None* under s' just as *none* does under s. The stack of the new frame is initially empty, hence we replace all *St* expressions with *none*. Temporal expressions also need to be modified. In our example we have $eval\ \Pi\ s'\ (Call\ (St\ 0)) = \lfloor Intg\ 3 \rfloor$. To obtain the same value in the previous state, we drop the *Call* operator, because *callstate* $s' = s$. In case of *Catch X ex* we check whether the current method has a catching handler. If so, we can eliminate the *Catch*. In Fig. 5.7 this is not the case, because *set* does not provide any handler at all. In Fig. 3.10 (p. 63) the *main* method, which

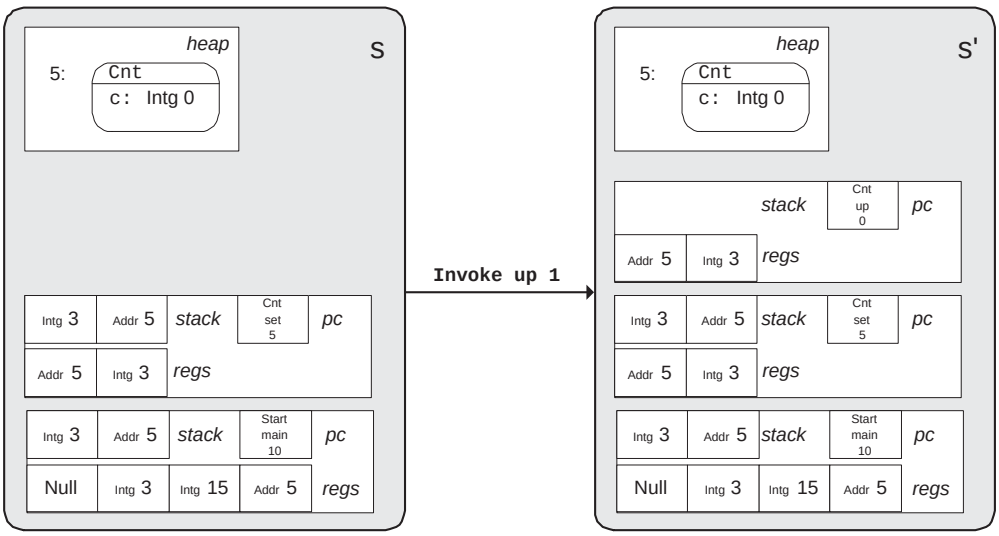


Figure 5.7: Invoking *up* from (*Cnt*,*set*,*5*).

has some handlers, calls *up*. In this situation *wpNrm* would transform *Catch No ex* to *ex*, because *evalE* Π *s′* (*Catch No ex*) = *evalE* Π (*catchstate* (Π,*No*,*s′*)) *ex* = *evalE* Π *s ex*. Removing *Catch* also works in situations, where we only have one frame on the stack. This is because we defined *catchstate* such that it becomes an identity in these situations. In all other situations, where we have at least two frames and no catching handler, we leave *Catch X*, because we have *catchstate* (Π,*s′*,*X*) = *catchstate* (Π,*s*,*X*). In Fig. 5.7 this applies, hence *wpNrm* leaves *Catch No* (*St 0*) as it is.

```
wpNrm (P, An) p (C′, M′, pc′) Q pm (Invoke M n) =
(let (_, _, (_, _, (mxl, _))) = method P C′ M
 in substE
    (pm @ (FrNr, FrNr +1 Intg 1)·
     map (λk. (Rg k, if k ≤ n then St (n − k) else if k ≤ n + mxl then arbitrary else none))
      (rgIds Q) @
     map (λk. (St k, none)) (stkIds Q) @
     map (λex. (Call ex, ex)) (getCallEx Q) @
     concat
      (map (λ(cn′, ex′).
              if catchesEx P cn′ p then [(Catch cn′ ex′, ex′)]
              else [(Catch cn′ ex′, if FrNr =1 Intg 1 then ex′ else Catch cn′ ex′)])
        (getCatchEx Q)))
    Q)
```

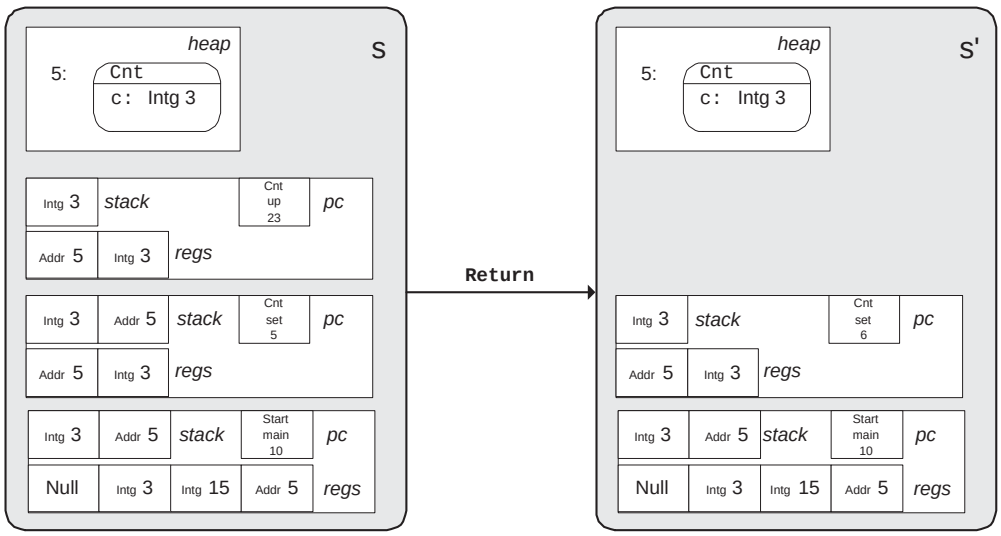


Figure 5.8: Returning from up

In case of `Return` the successor state has one frame less. Hence, the evaluation of *Call* and *Catch* expressions needs to be adjusted again. Adding an additional *Call* to such expressions amounts to the same as chopping off the topmost frame of the current state.

For example in Fig. 5.8 we illustrate how *up* returns to *set*. If we evaluate *Call* (*Rg 0*) in state s' we obtain $\lfloor Null \rfloor$. To get the same result in state s we have to add one *Call* and evaluate *Call* (*Call* (*Rg 0*)). The reduction of the frame stack can be compensated by replacing *FrNr* with *FrNr* $\llcorner\!\lrcorner$ $\lfloor Intg\ 1 \rfloor$. Note that execution ends, when the main method returns. Our semantics *effS* and control flow function *succsF* do not yield successors in this situation. Hence, whenever we apply *wpF* on a `Return`, there are at least two entries on the frame stack. To hand back the result value `Return` pushes the topmost value from the stack onto the stack of the caller. That means *St 0* evaluates to the same value before and after. For the remaining *St k* expressions, we have to switch the context using *Call*. The same needs to be done with registers.

```
wpNrm (P, An) (C, M, pc) p' Q pm Return =
(let (_, aTys, _) = method P C M
 in substE
     (pm @ (FrNr, FrNr ⌐¬ ⌞Intg 1⌟) ·
      map (λk. (St k, if 1 ≤ k then Call (St (|aTys| + k)) else St 0)) (stkIds Q) @
      map (λk. (Rg k, Call (Rg k))) (rgIds Q) @
      map (λex. (Call ex, Call (Call ex))) (getCallEx Q) @
      map (λ(cn', ex'). (Catch cn' ex', Call (Catch cn' ex'))) (getCatchEx Q))
     Q)
```

Exceptional Weakest Preconditions

Exception handling works quite similar for all instructions. Only `Throw` needs to be treated separately, because it either throws the reference lying on top of the stack or a *NullPointer* exception. We handle this diversity by replacing *St 0* with a conditional expression. If the reference is not *Null*, we keep it, otherwise we replace it with the address of the pre-allocated system exception *NullPointer*. This reference is the only thing lying on the stack before entering the handler, all other entries are removed. Hence, we replace all other stack expressions with *none*. If the instruction is not `Throw`, the thrown exception can only be a system exception. We use the function *sys-xcpt-of* to determine the kind of system exception from the instruction *i*.

```
sys-xcpt-of :: instr ⇒ cname
sys-xcpt-of i =
case i of New C ⇒ OutOfMemory
| Getfield F C ⇒ NullPointer
| Putfield F C ⇒ NullPointer
| Checkcast C ⇒ ClassCast
| Invoke M n ⇒ NullPointer
| Throw ⇒ NullPointer
| - ⇒ Exception
```

Then we replace *St 0* with the address of the corresponding pre allocated system exception. Next, we have to handle *Catch* and *Call* expressions. We can check statically if the thrown exception is caught inside the current method. If so, the frame stack stays the same and temporal expressions must not be modified. Otherwise, we pack frame dependent expressions into a *Catch cn* operator, which we introduced in particular for this situation. We only have to do this for register expressions, other temporal expressions and *FrNr*. Other expressions are either compositions, which only need to modified at

their leaves, are already handled, like *Pos*, or are not affected by changes on the frame stack, like heap expressions.

wpExc:: *jbc-prog* ⇒ *pos* ⇒ *pos* ⇒ *expr* ⇒ (*expr* ∼∼> *expr*) ⇒ *cname* ⇒ *instr* ⇒ *expr*

```
wpExc Π p p' Q pm cn i = (let mp=pm@
(map (λk. (St k,
     (if 1≤k then none
     else (if i = Throw
          then (⌊if St 0 ≔ ⌊Null⌋
               ⌊then⌋ ⌊Addr (addr-of-sys-xcpt NullPointer)⌋
               ⌊else⌋ St 0)
          else ⌊Addr (addr-of-sys-xcpt (sys-xcpt-of i))⌋)))))
   (stkIds Q))@
(let (C,M,pc)=p; (C',M',pc')=p'; (P,An)=Π
 in (if match-ex-table P cn pc (ex-table-of P C M) = ⌊pc'⌋
     then []
     else let
      rgm= map (λk. (Rg k,Catch cn (Rg k))) (rgIds Q);
      om = map (λex. (Call ex,Catch cn (Call ex))) (getCallEx Q);
      cm = map (λ(cn',ex'). (Catch cn' ex', Catch cn (Catch cn' ex')))
             (getCatchEx Q)
        in (FrNr,Catch cn FrNr) · rgm@om@cm))
                    in substE mp Q)
```

5.3 Conclusion

In this chapter we defined the main workhorses for our VCG. The control flow function *succsF* and the abstract semantics in form of *initF* and *wpF*. These functions embody the major complexity of a VCG and demonstrate that having a non-verified VCG is a high risk. In fact, we needed to debug our definitions several times until we could prove the requirements in §6.6 and instantiate our correctness and completeness theorems in §6.7. Our control flow function and abstract semantics operate on formulas of the safety logic we defined in §4, and are thus heavily influenced by the design choices we made there. Since we chose to embed safety formulas deeply, we are able to extract type information from annotations and can use it to narrow the potential successors for dynamic method calls or exceptions. The same technique could be applied to determine the successors of bytecode subroutines [56, 94]. Being able to distinguish formulas syntactically also allows us to distinguish formulas that access the heap. The *wpF* function knows that a `Putfield` *F C* instruction cannot affect expressions fetching fields from different classes or with different names. This helps to localise changes in the heap and offers similar benefits than the split heap approach [26, 24, 61]. In Necula's approach [68] the heap

is a variable r_m and the assertion language has constructs for dynamic heap update and selection. In case of `Putfield` F C the symbolic evaluator would simply replace r_m with $upd\ r_m\ (St\ 0)\ (St\ 1)$ and the logic would resolve heap changes with special update rules, e.g. $sel\ (upd\ x\ y\ z)\ y' = (\mathit{if}\ y = y'\ \mathit{then}\ z\ \mathit{else}\ sel\ x\ y')$. As one can see this eventually also boils down to conditional expressions. However, the variable r_m has a function type, whereas our assertions always evaluate to primitive Jinja values. The downside of using a deep embedding is that we have to use complicated substitutions to define *wpF*. If we used a shallow embedding, we would define formulas as predicates, i.e. $Q :: jbc\text{-}state \Rightarrow bool$, and could define *wp* in one line:

$$wp\ p\ p'\ \Pi\ Q = \lambda s.\ (\forall s'.\ (fst\ s' = p' \wedge ((p,snd\ s),(p',snd\ s')) \in set\ (effS\ \Pi)) \longrightarrow Q\ s')$$

Although this would also greatly simplify proving our framework's requirements, this approach has disadvantages. For example, *wp* does not abstract from the concrete semantics *effS*. This means state effects Q is not interested in would also appear in s' and thus be mentioned in the resulting verification condition. For example, assume we have `Push` $(Intg\ 3)$ at position p, i.e. $cmd\ \Pi\ p = \lfloor$`Push` $(Intg\ 3)\rfloor$, and a postcondition Q that is only interested in register 0, e.g. $Q = (Rg\ 0 \sqsupseteq \lfloor Null \rfloor)$. With *wpF* we obtain the following precondition:

$$wpF\ \Pi\ p\ p'\ Q = substE\ []\ Q = (Rg\ 0 \sqsupseteq \lfloor Null \rfloor)$$

Nothing changes, because no affected expression can be extracted from Q!

In the shallow embedding we could express Q as $Q_s = (\lambda s.\ rg\ 0\ s = Null)$, where *rg* is a function fetching registers from a state. Now, if we apply the shallow weakest precondition operator *wp*, we obtain the following:

$$wp\ p\ p'\ \Pi\ Q_s = (\lambda s.\ (\forall s'.\ (fst\ s' = p' \wedge ((p,snd\ s),(p',snd\ s')) \in (effS\ \Pi)) \longrightarrow (\lambda s.\ rg\ 0\ s = Null)\ s'))$$

Schirmer [83] effectively demonstrates that one could write Isabelle/HOL tactics that apply beta reduction, lookup $cmd\ \Pi\ p$ and resolve the effect of *effS* to beautify a shallow formula. However, all the steps of the tactic result in proof steps and must be recorded, sent to and checked by the consumer. We cannot directly send the small, beautified formula, because the consumer can only reproduce the raw output of the VCG. From an earlier instantiation [96], where we directly compare a deep with a shallow assertion logic, we made the following experience: In a shallow embedding the definition of the "abstract" semantics is simple and elegant, but the resulting verification conditions are complex (HOL) and big. In a deep embedding the functions are complex (substitution), but the resulting formulas simple (FOA) and small.

6 Verification Conditions for Jinja

This chapter defines the remaining parameters for our VCG and instantiates it by putting the pieces together. We also show examples of verification conditions and explain why these are modular. To carry over the theorems from the framework to our instantiation, we have to prove that the parameters satisfy the requirements. These proofs are the main effort in instantiations and we discuss some of them in detail.

6.1 SafetyPolicy

Our VCG expects the safety policy to be defined via local safety formulas. We have to define a function *safeF*, which yields a safety formula for every position in a program Π.

safeF :: *jbc-prog* ⇒ *pos* ⇒ *expr*

In our case, we specify a policy against arithmetic overflow. In Java the highest positive integer is *maxI*, i.e. $maxI = 2^{31}-1 = 2147483647$. If the result of an arithmetic operation lies above that number, Java silently overflows. In our safety logic we have normal integers and $\lfloor + \rfloor$ means the ordinary unbounded addition on integers. Hence, we can express the situation of an overflow by computing the real result in the integers and checking whether it lies within the representable region. For simplicity, we do not check underflow here.

safeF Π *p* = (*case cmd* Π *p of None* ⇒ $\lfloor F \rfloor$ | $\lfloor i \rfloor$ ⇒
(*case i of* IBin *no* ⇒ *Num* (*St 1*) *no* (*St 0*) $\leqslant$ $\lfloor$*Intg 2147483647*$\rfloor$
| - ⇒ $\lfloor T \rfloor$))

In Jinja only the binary arithmetic operation can overflow. If *p* points to such an operation we specify a condition that computes the real result and checks whether it is low enough. If *p* is outside the code domain, *safeF* Π *p* yields $\lfloor F \rfloor$. This means programs that can reach such a position are unsafe. Positions with other instructions are always safe to execute; we express this with the trivial safety formula $\lfloor T \rfloor$.

6.2 Wellformedness

Some of our VCG's parameter functions require Jinja programs to satisfy a few easily checkable wellformedness constraints. In this section we specify these and discuss why they are helpful or even necessary. We call a program Π wellformed, i.e. *wf* Π, if and only if

(1) control flows within the code domain and there are enough annotations,

(2) exception handlers do not overlap and only expect one argument on the operand stack (reference to exception),

(3) all classnames and method names per class have distinct names,

(4) system classes are declared,

(5) the position (*Start*,*main*,*0*) exists,

(6) they are accepted by the Jinja bytecode verifier, and

(7) the *main* method has no arguments.

Below is the formal definition of these properties. We discuss some of the checks below. For the sake of brevity, we skip the formal definitions of auxiliary functions used here. Interested readers find the full definitions in the appendix §A.2.2 or otherwise in the Jinja article [55].

wf :: *jbc-prog* ⇒ *bool*

wf Π = (*checkPos* Π (*domC* Π) ∧
checkExTables Π ∧
distinct (*classnames* (*fst* Π)) ∧ *distinct* (*methodnames* (*fst* Π)) ∧
(∃ *cdl*. *fst* Π = (*SystemClasses* @ *cdl*)) ∧
(*ipc* Π ∈ *set* (*domC* Π)) ∧
wf-jvm-prog-phi (*map-of2* (*convert-pt* (*prog-kil* (*fst* Π)))) (*fst* Π) ∧
fst (*snd* (*method* (*fst* Π) (*fst* (*ipc* Π)) (*fst* (*snd* (*ipc* Π))))) = [])

Comment on 1
We check for each program position *p*, that all its normal and exceptional successors lie within the code domain *domC* Π and are distinct. The first is required for the completeness proof and an essential requirement for safety anyway. The latter enables our *wpF* operator to distinguish normal from exceptional execution by checking whether control flows into a handler. We also check that targets of backward jumps have annotations.

This ensures that there are no cycles without annotations, which would prevent our VCG from terminating. Moreover `Throw` and `Invoke` instructions must have type annotations. Types are extracted by *succsF* to narrow down possible targets. All these properties are checked by *checkPos* Π (*domC* Π), whose formal definition is in the appendix §A.2.2.

Comment on 2
For all exception handler tables *xt* we demand that its entries (f,t,C,h,d) have $d{=}0$ and unique entry positions h. With $d{=}0$ we say that exception handlers may only be entered with an operand stack that contains nothing, but the reference to the exception. In [55] exceptions may be thrown and caught during expression evaluation and d additional stack arguments may be taken by the handler. Demanding unique entry positions means that if we have another handler (f',t',C',h',d') ∈ *set xt*, then $h' \neq h$. The uniqueness property is just a simplifiction that allows us to define *handlesEx*, which is used by *wpF*, as a function. Note that both conditions automatically hold for all bytecode programs compiled from Java sources. In Java, exceptions are only caught at the statement level ($d{=}0$) and handlers are unique as compilers do not share handlers with common code.

Comment on 3
We require that all classnames and all methods per class have distinct names. This guarantees that *domC*, which recurses over all class declarations and methods therein, does not yield dead positions. We call positions dead if they belong to methods that are not obtainable by the *method* lookup function, because they coincide with other entries having higher priority in the class map (represented as list). Both restrictions could be dropped, but this would complicate our definition of *domC*.

Comment on 4
The system classes *Exception*, *ClassCast*, *OutOfMemory*, *NullPointer* and *Object* must be declared in the program. This is important, because otherwise the method and field lookup functions do not work properly.

Comment on 5
The initial program position *ipc* Π, which is (*Start*,*main*,*0*) must exist in the code domain. This is ensures requirement 2.12 of our framework.

Comment on 6
Programs accepted by the Jinja bytecode verifier are type safe. This means our semantics *effS* never gets stuck, because operands do not have the expected format. Since [55] provides an Isabelle/HOL proof for type safety, we can trust the bytecode verifier and

use its types in §6.4 to upgrade the branch conditions of our successor function.

Comment on 7
The type safety theorem for Jinja bytecode [55] assumes that the *main* method has no arguments. This restriction is not essential, but simplifies the specification of initial states. Since we rely on type safety in order to prove lemma 6.4, we also demand this condition.

6.3 System Invariants

In §3.2 we model states of the Jinja virtual machine as the HOL type *jbc-state*. Although this type already gives a detailed picture of how states are structured, it is just a coarse definition of states a Jinja program can actually have. Many elements of this type are "states" that will never occur in any running Jinja program. The operational semantics and the wellformedness constraints assure certain properties every reachable state satisfies automatically.
For example, in every state of a wellformed program the program counter points to some instruction and all operands have the type this instruction expects. We call such properties *system invariants* as the system guarantees them for every wellformed program.
To prove verification conditions it is sometimes necessary to explicitly express such system invariants in the safety logic. Making these invariants a part of annotations is one way to reach this goal. From the logical point of view this is perfectly acceptable, but it comes at a price. Annotations need to be verified! Although many system invariants are properties that could be verified automatically, they usually cause a lot of unnecessary clutter in a safety proof.
A better way to deal with such properties is to prove them once and for all as invariants that hold for any program and then make them available in proofs as trusted facts. Our framework allows to do this by packing system invariants into branch conditions, which only appear at the left hand side of implications in the proof obligations.
Another reason why system invariants are interesting is that they are very helpful in the verification of the requirements our PCC framework poses on various parameter functions. For this reason, we now introduce a few essential properties, which we have proven to be system invariants. We represent these system invariants as functions *inv-...* yielding a formula in the safety logic for every program position.

inv-... :: *jbc-prog* $\Rightarrow$ *pos* $\Rightarrow$ *expr*

System invariant *inv-Pos* follows from the constraints *wf* Π poses on the control flow.

inv-Pos Π p = *Pos* p

Although *inv-Pos* essentially is just a different name for *Pos*. We introduce it to make clear that it is a system invariant and to make its signature compatible to other system invariants that may depend on the additional parameter Π. Note that *Pos p* not only fixes the program counter to *p*, but also guarantees the call stack to be wellformed. The following lemma says that *inv-Pos* holds for all reachable states in wellformed programs. In other words, it is a system invariant.

Lemma 6.1 The predicate *inv-Pos* covers all reachable states.

$wf\ \Pi \wedge s \in Reachables\ \Pi \longrightarrow \Pi, s \models inv\text{-}Pos\ \Pi\ (fst\ s)$

Control flow safety also guarantees that every `Return` instruction, except for the terminating one from the *main* method, finds at least two frames on the frame stack. This property is quite important for many proof rules we derive for our *Call* and *Catch* operators.

$$inv\text{-}FrNr\ \Pi\ (C,M,pc) = (\mathsf{if}\ C{=}Start \wedge M{=}main\ \mathsf{then}\ FrNr \leftrightharpoons \lfloor Intg\ 1 \rfloor \ \mathsf{else}\ (\lfloor Intg\ 1 \rfloor \lesssim FrNr))$$

Lemma 6.2 The predicate *inv-FrNr* covers all reachable states.

$wf\ \Pi \wedge s \in Reachables\ \Pi \longrightarrow \Pi, s \models inv\text{-}FrNr\ \Pi\ (fst\ s)$

Another property of all reachable Jinja states is that pre-allocated exception objects remain in the heap at specific addresses.

$$inv\text{-}ExTys\ \Pi\ p = \bigwedge [Ty\ \lfloor Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ NullPointer) \rfloor\ (Class\ NullPointer),\ Ty\ \lfloor Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ ClassCast) \rfloor\ (Class\ ClassCast),\ Ty\ \lfloor Addr\ (addr\text{-}of\text{-}sys\text{-}xcpt\ OutOfMemory) \rfloor\ (Class\ OutOfMemory)]$$

Lemma 6.3 The predicate *inv-ExTys* covers all reachable states.

$wf\ \Pi \wedge s \in Reachables\ \Pi \longrightarrow \Pi, s \models\ inv\text{-}ExTys\ \Pi\ (fst\ s)$

Apart from the system exceptions, we can also construct annotations constraining the types of registers and operands from the bytecode verifiers type inference.

$inv\text{-}Ty\ \Pi\ p = annotate\text{-}types\ (fst\ \Pi)\ (convert\text{-}pt\ (prog\text{-}kil\ (fst\ \Pi)))\ p$

The Jinja bytecode verifier, i.e. *prog-kil*, infers types for registers and stack elements. Converted to our assertion language, i.e *annotate-types*, these facts become expressions of the form $Ty\ (Rg\ 0)\ (Class\ Start) \wedge \ldots \wedge Ty\ (St\ 0)\ Integer \wedge \ldots$.
In §7.1.1 we elaborate more on the bytecode verifier (BCV). In [55] one finds all the details about it, including a proof of its correctness. Types inferred by *prog-kil*, that are accepted by the type checker *wt-jvm-prog-kil*, are guaranteed to hold at runtime. Note that our wellformedness checker *wf* uses this type checker to enforce welltypedness. From

the BCV's correctness and this check, we can then derive the following lemma, which says that *inv-Ty* is a system invariant.

Lemma 6.4 Function *inv-Ty* is a system invariant.
$wf\ \Pi \wedge s \in Reachables\ \Pi \longrightarrow \Pi, s \models\ inv\text{-}Ty\ \Pi\ (fst\ s)$

6.4 Instantiating the VCG

Having defined all parameter functions, we can now instantiate our generic VCG to Jinja. We do this in two steps. First, we define a function *vcg* using our global operator *vcG* taking all parameters and combining them to a verification condition generator. Note that we also have used *vcG* in our locale VCG, but only have shown derived lemmas from that. The reason is that the definition of *vcG* is quite unreadable due to its many parameters and due to an artifical list of positions used to enforce termination even for non-wellformed programs. In §2.6 we rather present derived equations for *vcg*. These can also be seen as definitions under the assumption of wellformed programs.

$vcg\ \Pi = vcG\ \bigsqcap\ \Rightarrow\ F\ ipc\ initF\ safeF\ succsF\ wpF\ domC\ domA\ anF\ \Pi$

We also instantiate other VCGs using upgraded control flow functions. Theorem 2.7 (p. 48) enables us to add invariants to branch condition. Since, we have proven that our system invariants are invariants, we will now upgrade them to *succsF*. For this purpose we use the upgrade function *upg* from §2.10. We start with *inv-FrNr*, which ensures that there are always enough frames on the stack.

$succsFrNrF :: jbc\text{-}prog \Rightarrow pos \Rightarrow (pos \times expr)\ list$

$succsFrNrF = upg\ inv\text{-}FrNr\ succsF$

Then, we upgrade *inv-ExTy*, which ensures that objects for system exceptions are allocated.

$succsExTysF :: jbc\text{-}prog \Rightarrow pos \Rightarrow (pos \times expr)\ list$

$succsExTysF = upg\ inv\text{-}ExTys\ succsFrNrF$

Finally, we upgrade the types the bytecode verifier infers. With *inv-Tys* we translate these to expressions.

$succsTysF :: jbc\text{-}prog \Rightarrow pos \Rightarrow (pos \times expr)\ list$

$succsTyF = upg\ inv\text{-}Ty\ succsExTysF$

Although *succsTyF* clutters up the branch conditions from *succsF* with lots of detailed type information, it has one important advantage. It guarantees progress as demanded

by requirement 2.11 from §2.8. In §6.7 this will play a vital role for our completeness proof.

Taking the upgraded control flow functions, we now define upgraded versions of our VCG.

$vcgFrNr\ \Pi = vcG\ \lfloor\bigwedge\rfloor\ \lfloor\Rightarrow\rfloor\ \lfloor F\rfloor\ ipc\ initF\ safeF\ succsFrNrF\ wpF\ domC\ domA\ anF\ \Pi$

$vcgExTys\ \Pi = vcG\ \lfloor\bigwedge\rfloor\ \lfloor\Rightarrow\rfloor\ \lfloor F\rfloor\ ipc\ initF\ safeF\ succsExTysF\ wpF\ domC\ domA\ anF\ \Pi$

$vcgTy\ \Pi = vcG\ \lfloor\bigwedge\rfloor\ \lfloor\Rightarrow\rfloor\ \lfloor F\rfloor\ ipc\ initF\ safeF\ succsTyF\ wpF\ domC\ domA\ anF\ \Pi$

With *vcgTy* we have arrived at a VCG that emits complete verification conditions. In §6.7 we will show the corresponding theorem. The reason why we also define and mention the other VCGs is that for practical purposes incomplete VCGs are often better. Note that with every upgrade the resulting verification conditions contains more facts. For some safety policies or programs many of these facts will not be needed to prove the verification condition. In this case they only lead to big verification conditions and safety proofs. Therefore it is practical to try to prove safety with a weaker VCG first. Only if the verification condition cannot be proven automatically, one should switch to more powerful VCGs. When one has arrived at *vcgTy* and the proof still cannot be constructed automatically, this is because either the program is unsafe, the annotations are wrong, or the proof procedures are insufficient. However, the completeness theorem guarantees that the reason is not the VCG, which may have missed some important facts or constructed verification conditions that are too restrictive.

6.5 Verification Conditions and Modularity

Java programmers keep code modular by distributing it among various methods. For practical reasons it is quite important that our VCG also respects this modularity. The proof for a method body must be strictly detached from its call contexts. Otherwise we would have to adjust annotations and proofs of methods every time we add or remove code that uses this method. Verified libraries would become impossible. In Hoare Logic modularity is achieved by working with modular specifications of methods, typically in form of pre- and postconditions. We follow this idea and require entry and exit positions of methods to be annotated with such conditions. Pre- and postconditions are not modular if they depend on a particular call context. To give an example, consider our program from Fig. 3.3. At position $(Start,main,14)$ the *main* method invokes *up* in order to increase the counter field c from $Intg\ x_0$ to $Intg\ (x_0 + y_0)$.

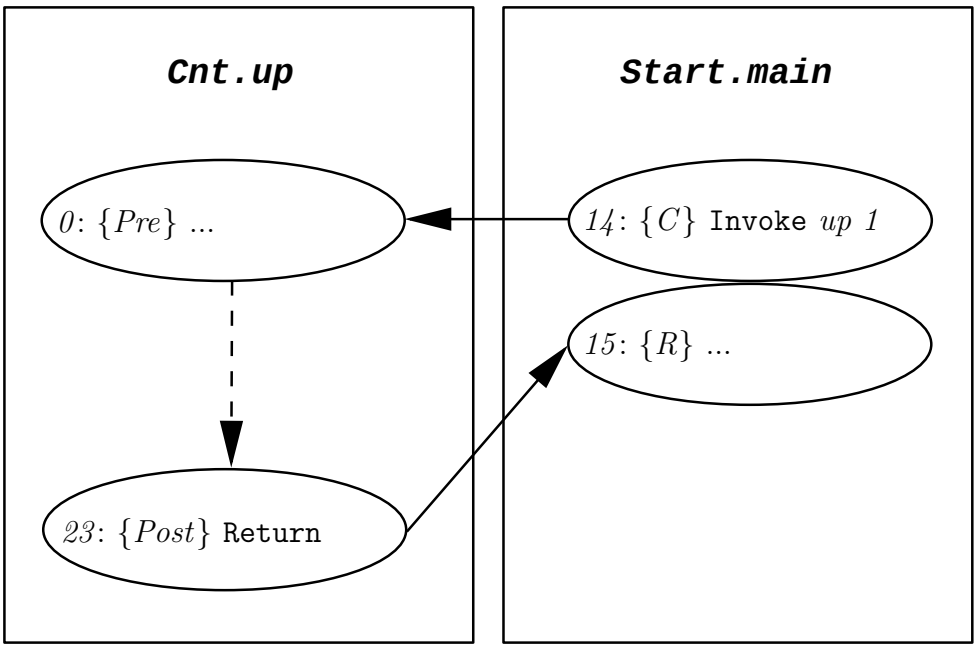


Figure 6.1: Method invocation and return

In Fig. 6.1 and Fig. 3.10 we illustrate the situation. Immediately before *up* becomes invoked, the machine is in a state, where:

- register 1 and 2 contain the integers *Intg* x_0 and *Intg* y_0 (x_0=*3*, y_0=*15*).
- register 3 points to a *Cnt* object, whose *c* field is set to *Intg* x_0.
- the argument *Intg* y_0 lies on top of the stack, followed by a copy of the reference from register 3.

The call condition C describes this situation formally. It is valid whenever we reach the call position (*Start*,*main*,*14*) at runtime, thus we can use it as annotation.

C = *aF* Π (*Start*,*main*,*14*) =
$\bigwedge$ [*Ty* (*St 1*) (*Class Cnt*), *Rg 1* ≕ ⌊*Intg* x_0⌋, *Rg 2* ≕ ⌊*Intg* y_0⌋, *Gf c Cnt* (*Rg 3*) ≕ ⌊*Intg* x_0⌋, *St 0* ≕ ⌊*Intg* y_0⌋, *St 1* ≕ *Rg 3*]

When *up* returns to *main* from position (*Cnt*,*up*,*23*) (see Fig. 3.11), the result *Intg* (x_0 + y_0) lies on top of the stack and the counter field *c* has been updated with it. All the other things are just as before. Again, we can formulate these expectations in a so called

return condition R and annotate it to position (*Start*,*main*,*15*).

$R = aF\ \Pi\ (Start,main,15) =$
$\bigsqcap [Rg\ 1 \sqsupseteq \lfloor Intg\ x_0 \rfloor, Rg\ 2 \sqsupseteq \lfloor Intg\ y_0 \rfloor, Ty\ (Rg\ 3)\ (Class\ Cnt), St\ 0 \sqsupseteq Gf\ c\ Cnt\ (Rg\ 3), Gf\ c\ Cnt\ (Rg\ 3) \sqsupseteq \lfloor Intg\ (x_0 + y_0) \rfloor]$

Together, call condition C and return condition R make up a call context for method *up*. Note that x_0 and y_0 are constants standing for the input values, e.g. *Intg 3* and *Intg 15*. If (C,R) is the only call context, we could verify method *up* with a precondition claiming $Rg\ 1 \sqsupseteq \lfloor Intg\ 3 \rfloor$ and a postcondition directly referring to the result value $\lfloor Intg\ 18 \rfloor$, i.e. $St\ 0 = \lfloor Intg\ 18 \rfloor$. Even if there are other call contexts we could disjunct the facts from the different call contexts to obtain valid pre and postconditions. However, such conditions would not be modular. If additional call contexts come into play, we have to adjust the method specification and its proofs. To achieve modular pre and postconditions, we follow the idea of VDM [53], which uses a temporal operator $'$ to interpret variables x in the initial state of a method, i.e. x'. In our case the *Call* operator serves this purpose. Below we have the formulas *Pre* and *Post*. They show how we can specify the behaviour of *up* in a modular way.

In *Pre*, we say that register 0 (`this`) contains a reference to *Cnt* object. Register 1 contains an integer within the representable range. These two registers are filled with the arguments the method has been invoked on. Using the *Call* operator we link them to the values lying on the operand stack at call time. Finally *Pre* claims that the object field c has not changed so far.

$Pre = aF\ \Pi\ (Cnt,up,0) =$
$\bigsqcap [Ty\ (Rg\ 0)\ (Class\ Cnt), Ty\ (Rg\ 1)\ Integer, Rg\ 1 \trianglelefteq \lfloor Intg\ 2147483647 \rfloor, Rg\ 0 \sqsupseteq Call\ (St\ 1), Rg\ 1 \sqsupseteq Call\ (St\ 0), Gf\ c\ Cnt\ (Rg\ 0) \sqsupseteq Call\ (Gf\ c\ Cnt\ (St\ 1))]$

The postcondition *Post* describes the result of *up* when no exception occurs, that is when we reach position (*Cnt*,*up*,*23*). It says that register 0 still contains a reference to a *Cnt* object and that this reference is the one passed as an argument. It also says that the object field c has been increased by the second argument and that the same value is lying on top of the stack, ready for being returned as result value.

$Post = aF\ \Pi\ (Cnt,up,23) =$
$\bigsqcap [Ty\ (Rg\ 0)\ (Class\ Cnt), Rg\ 0 \sqsupseteq Call\ (St\ 1), St\ 0 \sqsupseteq Gf\ c\ Cnt\ (Rg\ 0), Gf\ c\ Cnt\ (Rg\ 0) \sqsupseteq (Call\ (St\ 0) \sqcup Call\ (Gf\ c\ Cnt\ (St\ 1)))]$

Now, we have to think about how to verify method *up* and its invocations. In Hoare Logic, one would have to show two sorts of triples. One for the method body, i.e. $\{Pre\}$ *body* $\{Post\}$, and one for each invocation, e.g. $\{C\}$ `Invoke` *up 1* $\{R\}$.

6.5.1 Verifying Method Bodies

To verify the body of method *up*, we simply have to annotate its entry position (*Cnt*,*up*,*0*) with *Pre* and its exit position (*Cnt*,*up*,*23*) with *Post*. When our VCG reaches the annotated position (*Cnt*,*up*,*0*) it automatically starts producing proof obligations similar to those of the Hoare triple for method bodies.

In our example *vcgFrNr* produces the proof obligation *vc-Cnt-up-0* for position (*Cnt*,*up*,*0*), which we show in Fig. 6.2. In essence, it is a big implication ensuring that from a state satisfying the precondition we can reach the exit position (*Cnt*,*up*,*23*) only in states satisfying the postcondition. In the example we find the precondition *Pre* in lines 1-2 and the postcondition *Post* (manipulated by *wpF*) in lines 40-45. For every control flow transition, we get an implication. On the left hand side we find the branch condition, on the right hand side the safety formula of the next position and the formula for the remaining path. In *up* there are only two critical operations, the subtraction at (*Cnt*,*up*,*7*) and the addition at (*Cnt*,*up*,*19*). The safety formulas in line 16 and 32 ensure that their result is below the highest integer (no overflow). All other safety formulas are simply $\lfloor T \rfloor$. For the branch conditions we use the abbreviations *P*, *F* and *Br*. In the real verification condition these stand for the following expressions:

$P = Pos\ (Cnt,\ up,\ 0)$
$F = \lfloor Intg\ 1 \rfloor \leq FrNr$
$Br = \bigwedge [\bigwedge [P,\lfloor T \rfloor],F]$

Each branch condition is a conjunct of three parts. First there are the position constraints, which are all assimilated to *P* by the weakest precondition operator. Then there is the target constraint, which ensures that we reach a particular successor. Most instructions do not throw exceptions or jump to various targets. Hence, we often find the trivial formula $\lfloor T \rfloor$ as target constraint. Finally, we have the frame stack constraint, which *vcgFrNr* adds by using *succsFrNrF*. The frame stack constraint is important, because it triggers our simplification rules for *Call* and *Catch* operators. In our example, the dynamic checks on the argument *Rg 1* result in the branch conditions on lines 10 and 22. Note that at these positions we have conditional jumps and the verification condition has obligations for both successors. However, in Fig. 6.2 we only show the parts for the normal execution of *up*. The placeholder __ indicates where the full verification condition has formulas ensuring that *up* also behaves correctly in case of exceptions. Although the formula in Fig. 6.2 looks complex, we can prove it automatically using Isabelle/HOL's combined tactic `(clarsimp,arith)`. It performs rewriting using our semantical proof rules, applies introduction and elimination rules and handles remaining numerical subgoals with arithmetic proof procedures. The branch conditions in Fig. 6.2 are mostly irrelevant. The optimised formula for *vc-Cnt-up-0*, which we show later in Fig. 7.4, visualizes the actual proof obligations much better.

```
vc-Cnt-up-0 = ⋀ [⋀ [⋀ [T, ⋀ [Ty (Rg 0) (Class Cnt), Ty (Rg 1) Integer,
  Rg 1 ≤ Intg 2147483647, Rg 0 = (Call (St 1)), Rg 1 = (Call (St 0)),
  (Gf c Cnt (Rg 0)) = (Call (Gf c Cnt (St 1)))]], Br]
 ⇒
 ⋀ [T, Br
   ⇒
  ⋀ [T, Br
    ⇒
   ⋀ [T, Br
     ⇒
    ⋀ [T, ⋀ [-- , ⋀ [P, ¬ (Rg 1 ≤ Intg 0), F]
      ⇒
     ⋀ [T, Br
       ⇒
      ⋀ [T, Br
        ⇒
       ⋀ [(Intg 2147483647 − Rg 1) ≤ Intg 2147483647, Br
         ⇒
        ⋀ [T, Br
          ⇒
         ⋀ [T, ⋀ [-- , ⋀ [P , ¬ (Rg 0 = Null)], F]
           ⇒
          ⋀ [T, ⋀ [--, ⋀ [P, (Intg 2147483647 − Rg 1) ≥ (Gf c Cnt (Rg 0))], F]
            ⇒
           ⋀ [T, Br
             ⇒
            ⋀ [T, Br
              ⇒
             ⋀ [T, Br
               ⇒
              ⋀ [T, Br
                ⇒
               ⋀ [(Gf c Cnt (Rg 0) + Rg 1) ≤ Intg 2147483647, Br
                 ⇒
                ⋀ [T, ⋀ [-- , ⋀ [P , ¬ (Rg 0 = Null)], F]
                  ⇒
                 ⋀ [T, Br
                   ⇒
                  ⋀ [T, ⋀ [-- , ⋀ [P , ¬ (Rg 0 = Null)], F]
                    ⇒
                   ⋀ [T, ⋀ [Ty (Rg 0) (Class Cnt), Rg 0 = (Call (St 1)),
  (if (Rg 0 = Rg 0) then (Gf c Cnt (Rg 0) + Rg 1)
  else (Gf c Cnt (Rg 0))) = (if (Rg 0 = Rg 0)
  then (Gf c Cnt (Rg 0) + Rg 1) else (Gf c Cnt (Rg 0))),
  (if Rg 0 = Rg 0 then (Gf c Cnt (Rg 0) + Rg 1) else Gf c Cnt (Rg 0))
  = ((Call (St 0)) + (Call (Gf c Cnt (St 1))))]]]]]]]]]]]]]]]]]]]]]
```

Figure 6.2: Verification Condition: Body `Cnt.up`

```
vc-main-up =
1  ⌊⋀⌋ [ ⌊⋀⌋ [⌊T⌋,
2  ⌊⋀⌋ [Ty (St 1) (Class Cnt), Rg 1 ⌐=¬ ⌊Intg x₀⌋, Rg 2 ⌐=¬ ⌊Intg y₀⌋,
3  (Gf c Cnt (Rg 3)) ⌐=¬ ⌊Intg x₀⌋, St 0 ⌐=¬ ⌊Intg y₀⌋, St 1 ⌐=¬ Rg 3]],
4
5  ⌊⋀⌋ [⌊⋀⌋ [Pos (Start, main, 14),
6  ⌊⋀⌋ [⌐¬ (St 1 ⌐=¬ ⌊Null⌋), Ty (St 1) (Class Cnt)]], FrNr ⌐=¬ ⌊Intg 1⌋] ]
7
8  ⌐⇒¬
9
10 ⌊⋀⌋ [⌊T⌋,
11 ⌊⋀⌋ [Ty (St 1) (Class Cnt), Ty (St 0) Integer, St 1 ⌐=¬ St 1,
12 St 0 ⌐=¬ St 0, (Gf c Cnt (St 1)) ⌐=¬ (Gf c Cnt (St 1))]]
13
14 vc-main-main =
15 ⌊⋀⌋ [ ⌊⋀⌋ [⌊T⌋,
16 ⌊⋀⌋ [Ty (St 1) (Class Cnt), Rg 1 ⌐=¬ ⌊Intg x₀⌋, Rg 2 ⌐=¬ ⌊Intg y₀⌋,
17 (Gf c Cnt (Rg 3)) ⌐=¬ ⌊Intg x₀⌋, St 0 ⌐=¬ ⌊Intg y₀⌋, St 1 ⌐=¬ Rg 3]],
18
19 ⌊⋀⌋ [⌊⋀⌋ [Pos (Start, main, 14),
20 ⌊⋀⌋ [St 1 ⌐=¬ ⌊Null⌋]], FrNr ⌐=¬ ⌊Intg 1⌋] ]
21
22 ⌐⇒¬
23
24 ⌊⋀⌋ [⌊T⌋,
25 ⌊⋀⌋ [Rg 1 ⌐=¬ ⌊Intg x₀⌋, Rg 2 ⌐=¬ ⌊Intg y₀⌋]]
26
27 vc-Start-main-14 = ⌊⋀⌋ [vc-main-up, vc-main-main]
```

Figure 6.3: Verification Condition: Invoking `Cnt.up`

6.5.2 Verifying Method Invocations

To verify method invocations $\{C\}$ `Invoke` $M\ n$ $\{R\}$, Hoare Logic gives us proof obligations of the form $C(x_1,\ldots,x_n) \longrightarrow Pre(x_1,\ldots x_n)$, where the x_i stand for the arguments. In our case we use *Rg* and *St* expressions for that purpose.

Our VCG gives us the formula *vc-main-up* shown in Fig. 6.3 for the discussed method invocation. We have to show the weakest precondition for *Pre* (lines 11-12) and can assume the call condition *C* (lines 2-3). Note that the precondition *Pre* links the arguments of the operand stack, about which *C* contains facts, with the registers inside the invoked method. In the goal (lines 11-12) the *wpF* operator has already resolved argument passing and we obtain trivial equations of the form *St 0* $\sqsubseteq$ *St 0*. The second formula *vc-main-main* treats the case when `Invoke` *up 1* fails due to a *NullPointer* ex-

ception and we end up in (*Start*,*main*,*23*). Note that the goal (line 25) follows trivially from the facts in line 16. This is an example of a formula the optimising function *opt*, which we will introduce in §7.3, reduces to $\lfloor T \rfloor$.

6.5.3 Verifying Method Returns

Finally, we have to show that *up* returns correctly to its caller. In Hoare Logic one typically uses a postcondition that relates the input values x_i with the output values y_i. Then one has to show that the return conditions holds for all y_i that are correct outputs of inputs satisfying C, i.e. $Post(x_1,\ldots,x_n,y_1,\ldots,y_n) \wedge C(x_1,\ldots,x_n) \longrightarrow R(y_1,\ldots,y_n)$.

When trying to adapt this proof obligation to our VCG, we run into difficulties. Note that the Hoare rule abstracts from the control flow, it goes directly from the call to the return condition. Since we chose to keep our VCG generic, it has to follow the control flow instead. For the method entry this is no problem. However, for the method exit, we end up with proof obligation of the form:

$\bigwedge \lfloor [Post,B] \Rightarrow wpF\ (Cnt,up,23)\ (Start,main,15)\ R \rfloor$

The problem here is that the return condition R depends on the call context, whereas *Post* must not in order to be modular. For example R claims that register 1 contains $\lfloor Intg\ x_0 \rfloor$, but *Post* does not mention this register at all. However, there is a neat way around this problem. We can pack call context specific information into branch conditions. Note that branch conditions are computed individually for each return position and need therefore not be modular. This means that we can take the call condition, wrap it into a *Call* operator in order to apply it to the registers, stack and heap at the call state, and pack it into B.

$B = \lfloor \bigwedge [\ldots,\ Call\ C,\ \ldots] \rfloor$

In Fig. 6.4 we show the verification condition we obtain for the normal exit of *up*, that is for position (*Cnt*,*up*,*23*). We only show the part for the return to (*Start*,*main*,*15*). The full verification condition also contains a formula for the return to (*Cnt*,*up*,*set*) using a completely different call context. We have to show the return condition R (lines 17-19) for states satisfying the postcondition *Post* (lines 2-4) and the branch condition B (lines 6-12). Note how the branch condition restores the call context by wrapping the call condition C with a *Call* operator (lines 7-11). Since, the frame stack is guaranteed to have at least two entries this call operator distributes inwards and we can directly establish many parts of the goal. For example in line 17 we have to show *Call* (*Rg 1*) $\Rightarrow \lfloor Intg\ x_0 \rfloor$, which we can obtain from line 8 after pushing the *Call* operator into the conjunction. Note that again `clarsimp` can solve this condition automatically.

```
vc-up-main =
1  ⋀ [ ⋀ [⊤,
2  ⋀ [Ty (Rg 0) (Class Cnt),
3  Rg 0 ⇀ (Call (St 1)), St 0 ⇀ (Gf c Cnt (Rg 0)),
4  (Gf c Cnt (Rg 0)) ⇀ ((Call (St 0)) ⊥ (Call (Gf c Cnt (St 1))))]],
5
6  ⋀ [⋀ [Pos (Cnt, up, 23),
7  Call (⋀ [⋀ [Ty (St 1) (Class Cnt),
8          Rg 1 ⇀ ↓Intg x₀, Rg 2 ⇀ ↓Intg y₀,
9          (Gf c Cnt (Rg 3)) ⇀ ↓Intg x₀,
10          St 0 ⇀ ↓Intg y₀, St 1 ⇀ Rg 3],
11          Pos (Start, main, 14)])],
12 ↓Intg 1 ≤ FrNr] ]
13
14 ⇒
15
16 ⋀ [⊤,
17 ⋀ [(Call (Rg 1)) ⇀ ↓Intg x₀, (Call (Rg 2)) ⇀ ↓Intg y₀,
18 Ty (Call (Rg 3)) (Class Cnt), St 0 ⇀ (Gf c Cnt (Call (Rg 3))),
19 (Gf c Cnt (Call (Rg 3))) ⇀ (↓Intg x₀ ⊥ ↓Intg y₀)]]
20
21 vc-Cnt-up-23 = ⋀ [ _ , vc-up-main]
```

Figure 6.4: Verification Condition: Return from `Cnt.up`

6.5.4 Exceptional Method Returns

So far we have ignored exceptions. The postcondition *Post* only specifies the normal behaviour of our method *up*. If one looks at the control flow graph in Fig. 5.1 there are many positions in *up* leading to one of the two exception handlers. All these positions can also be annotated with a postcondition stating what holds in case of an exception. For example we can annotate position (*Cnt*,*up*,*14*) with the following exceptional postcondition *PostE*, which roughly corresponds with the signals clause of the JML assertion shown in Fig. 3.3.

```
PostE =  ⋀ [Ty (St 0) (Class No), Rg 0 ⇀ (Call (St 1)), Rg 1 ⇀ (Call (St 0)),
(Gf c Cnt (Rg 0)) ⇀ (Call (Gf c Cnt (St 1))),
⋁ [(Rg 1) ≤ (Cn (Intg 0)), (↓Intg 2147483647 ⌐ (Rg 1)) ≤ (Gf c Cnt (Rg 0))]]
```

As Fig. 5.1 shows there are four edges leading out of (*Cnt*,*up*,*14*). For each of those we get different branch conditions and thus we end up with four different proof obligations for this position.

```
vc-Cnt-up-14-1 =
1  ⋀ [⋀ [Bool True, PostE],
2          ⋀ [⋀ [Pos (Cnt, up, 14),
3          ⋀ [Catch No (⋀ [Ty (St 1) (Class Cnt),
4                          Ty (Gf c Cnt (St 1)) Integer,
5                          Rg 1 ≔ Intg 3, Rg 2 ≔ Intg 15,
6                          St 0 ≔ Rg 1, St 1 ≔ Rg 3]),
7                Catch No (Pos (Start,main,10)),
8                Ty (St 0) (Class No)]],
9        Intg 1 ≤ FrNr]]
10
11 ⇒
12
13 (⋀ [Bool True,
14  ⋀ [(Catch No (Rg 1)) ≔ Intg 3, (Catch No (Rg 2)) ≔ Intg 15]])
15 vc-Cnt-up-14 = ⋀ [vc-Cnt-up-14-1, vc-Cnt-up-14-2, vc-Cnt-up-14-3, vc-Cnt-up-14-4]
```

Figure 6.5: Verification Condition: Exceptional return from `Cnt.up`

In Fig. 6.5 we only show one of the four formulas in detail, namely *vc-Cnt-up-14-1*. This formula handles the case when *up* is called from *set* with an improper argument. For example if we set x_0 to -3 the situation depicted in Fig. 4.2 occurs. Control flows to position (*Cnt*,*up*,*14*) and the created *No* exception is passed to the handler at (*Start*,*main*,*18*). Assume position (*Cnt*,*main*,*18*) is annotated with ⋀ [*Rg 1* ≔ *Intg* x_0, *Rg 2* ≔ *Intg* y_0, just as the JML assertion in Fig. 3.3 suggests. Then the *wpF* just wraps each register expression with *Catch No* (line 14). The successor function restores the call context, by wrapping the call condition also with *Catch No*. The verification condition can be proven automatically, because our simplification lemmas for ⊨ and *evalE* push the *Catch* operator inwards. Eventually, `clarsimp` finishes the goal from the facts in line 5. Note that `clarsimp` can also prove all the cases we have not shown here.

6.6 Proving Requirements

Apart from defining all the parameter functions, we also have to show that our definitions satisfy all the requirements our abstract framework demands. From the experience with various instantiations we can say that this boils down to verifying that *succsF* and *wpF* operate as expected. All the other requirements are usually trivial to verify.

6.6.1 Control Flow Approximation

For the correctness proof of our VCG we have to show that *succsF* does not forget any successors *effS* yields and that the branch conditions hold.

Lemma 6.5 Function *succsF* approximates the control flow and yields valid branch conditions.

$$\frac{wf\ \Pi \qquad s \in ReachablesAn\ \Pi \qquad (s,\ s') \in effS\ \Pi}{\exists B.\ (fst\ s',\ B) \in set\ (succsF\ \Pi\ (fst\ s)) \wedge \Pi,s \models\ B}$$

Proof To prove lemma 6.5, which discharges requirement 2.9, we first distinguish two cases: normal and exceptional execution of *effS*. In both cases, a lot of further case splits follow. In particular we do a case split on the kind of instruction the program counter of s points to. For each instruction, we apply the corresponding definition of *succsNrm* and show the goal by simplification. In the cases with exceptional execution, we have a separate lemma, which connects the outcome of *find-handler* with *succsXpt*. □

For the completeness proof we have to show that the branch conditions guarantee progress.

Lemma 6.6 Branch conditions of *succsF* ensure progress for states satisfying system invariants.

$$\frac{\begin{array}{c}\Pi = (P,\ An) \qquad p = (C,\ M,\ pc) \\ s = (p,\ (None,\ h,\ (st,\ rg,\ p)\cdot frs),\ e) \qquad wf\ \Pi \qquad (p',\ B) \in set\ (succsF\ \Pi\ p) \\ \Pi,s \models\ B \qquad cmd\ \Pi\ p = \lfloor i \rfloor \qquad \Pi,s \models\ inv\text{-}Pos\ \Pi\ p \\ \Pi,s \models\ inv\text{-}FrNr\ \Pi\ p \qquad \Pi,s \models\ inv\text{-}ExTys\ \Pi\ p \qquad \Pi,s \models\ inv\text{-}Ty\ \Pi\ p\end{array}}{\exists st'\ rg'\ frs'\ e'.\ (s,\ p',\ (st',\ rg',\ frs'),\ e') \in effS\ \Pi}$$

Proof To prove lemma 6.6 we first make a case distinction on the kind of instruction. For each instruction we separate the cases where p' is a normal or exceptional successor. In each case the branch condition restricts the state s such that either normal execution is possible or an exception occurs. In case of an exception we have that $(p',B') \in succsXpt\ (\Pi,X,[p])$. We generalise this to $succsXpt\ (\Pi,X,L)$, where $p = last\ L$, and start an induction on the difference $length\ (domC\ \Pi) - length\ L$, which becomes smaller with every recursive call of *succsXpt*. Then we instantiate L with $[p]$ and finish the proof. □

From lemma 6.6 we can derive two further lemmas, which establish the requirements 2.20 for *succsF* and 2.11 for *succsTyF*. All that needs to be done is to discharge the system invariants in lemma 6.6.

Lemma 6.7 For reachable states *succsF* yields branch conditions that ensure progress.

$$\frac{(p,\ \sigma,\ e) \in \mathit{Reachables}\ \Pi \qquad \overset{\mathit{wf}\ \Pi}{\Pi,(p,\ \sigma,\ e) \models\ B} \qquad (p',\ B) \in \mathit{set}\ (\mathit{succsF}\ \Pi\ p'')}{p = p'' \wedge (\exists \sigma'\ e'.\ ((p,\ \sigma,\ e),\ p',\ \sigma',\ e') \in \mathit{effS}\ \Pi)}$$

Proof Lemma 6.7 follows from lemma 6.6, because we can obtain the system invariants from $(p,\sigma,e) \in \mathit{Reachables}\ \Pi$. □

Lemma 6.8 The branch conditions of *succsTyF* guarantee progress.

$$\frac{\mathit{wf}\ \Pi \qquad \Pi,(p,\ m,\ e) \models\ B \qquad (p',\ B) \in \mathit{set}\ (\mathit{succsTyF}\ \Pi\ p'')}{p = p'' \wedge (\exists m'\ e'.\ ((p,\ m,\ e),\ p',\ m',\ e') \in \mathit{effS}\ \Pi)}$$

Proof We obtain lemma 6.8 from lemma 6.6 and the fact the *succsTyF* establishes all system invariants in its branch condition B.□

6.6.2 Abstract and Concrete Semantics

The symbolic evaluation of programs performed by our VCG must mimic the real behaviour of the Jinja VM. For the correctness of our VCG it suffices that the weakest precondition guarantees the postcondition in the successor state. Requirement 2.8 from §2.7, expresses this formally. It essentially demands the following implication:

$\ldots \wedge (s,s') \in (\mathit{effS}\ \Pi) \longrightarrow \Pi,s \models \mathit{wpF}\ \Pi\ (\mathit{fst}\ s)\ (\mathit{fst}\ s')\ Q \longrightarrow \Pi,s' \models Q$

This specifies that the formulas *wpF* produces must be strong enough, they must constrain s such that execution can only proceed with states s' satisfying Q. Note that this property does not prevent us from using a *wpF* function that is too strong. For example the function returning $\lfloor F \rfloor$ would perfectly satisfy this requirement. Hence, requirement 2.14 in 2.8 requires *wpF* to produce weakest preconditions only. For completeness the implication from above is turned around:

$\ldots \wedge (s,s') \in (\mathit{effS}\ \Pi) \longrightarrow\ \Pi,s' \models Q \longrightarrow \Pi,s \models \mathit{wpF}\ \Pi\ (\mathit{fst}\ s)\ (\mathit{fst}\ s')\ Q$

Instead of proving both requirements separately, we prove them at once by turning $\longrightarrow$ and $\longleftarrow$ into $=$. Since *wpF* directly operates on expressions, we prove that *wpF* preservers the evaluation of all expressions. In Fig. 6.6 we visualise this relationship.

$$\frac{\begin{array}{ccc} s & \xrightarrow{\mathit{effS}\ \Pi} & s' \\ Q' & \xleftarrow{\mathit{wpF}\ \Pi\ (\mathit{fst}\ s)\ (\mathit{fst}\ s')} & Q \end{array}}{\mathit{evalE}\ \Pi\ s'\ Q = \mathit{evalE}\ \Pi\ s\ Q'}$$

Figure 6.6: Abstract and Concrete Semantics

Note that for the correctness and completeness proof preserving validity ($\models$) of expressions would be enough. However, proving equivalence under evaluation (*evalE*) turns out to be more practical for induction on expressions. The lemma we actually prove makes a lot of assumptions, which we previously suppressed with .... Now, it is time to reveal the details:

Lemma 6.9 *wpF* mimics *effS*.

$$\frac{\begin{array}{c} \mathit{wf}\ \Pi \qquad s = (p,\ \sigma,\ e) \qquad s' = (p',\ \sigma',\ e') \qquad \Pi,s \models\ \mathit{inv\text{-}Pos}\ \Pi\ (\mathit{fst}\ s) \\ \Pi,s \models\ \mathit{inv\text{-}ExTys}\ \Pi\ (\mathit{fst}\ s) \qquad \Pi,s \models\ \mathit{inv\text{-}Ty}\ \Pi\ (\mathit{fst}\ s) \\ (p',\ B) \in \mathit{set}\ (\mathit{succsF}\ \Pi\ p) \qquad \Pi,s \models\ B \qquad (s,\ s') \in \mathit{effS}\ \Pi \end{array}}{\forall I.\ \mathit{evalE}\ \Pi\ (p,\ \sigma,\ e(\!|\mathit{lv} := I|\!))\ (\mathit{wpF}\ \Pi\ p\ p'\ Q) = \mathit{evalE}\ \Pi\ (p',\ \sigma',\ e'(\!|\mathit{lv} := I|\!))\ Q}$$

Proof Apart from wellformedness, which is a prerequisite for many other lemmas, the proof of lemma 6.9 requires the system invariants. These ensure that s has a valid frame stack, allocated exception objects and arguments of proper type on the operand stack and in the registers. We also assume that that the successor position p' lies in *succsF* Π p and that the branch condition B holds. From that and *wf* Π we can deduce that p' and p lie in *domC* Π and that *handlesEx* Π $p' = \mathit{None}$ in case of normal execution of *effS* Π. Normal or exceptional execution of *effS* Π is an early case split we make in the proof. In addition we make case distinctions on the instruction at p. For each instruction we then induct on the structure of Q. Since we have 16 instructions and 19 different expressions, we end up with 304 cases for the normal behaviour only. Since *wpF* treats exception handling uniformly, we only have one induction (19 cases) for the exceptional behaviour. Most cases, in particular the ones with composed expressions can be trivially handled by instantiating the induction hypotheses for the subexpressions. Difficult are the cases for `Putfield` F C, `New`, `Invoke`, `Return` and `Throw`. The first two modify the heap and we have to show the correctness for the substitutions of *Gf* F C *ex*

and *Ty ex tp* expressions. In the appendix §A.3.1 we have a detailed and commented Isar proof for the `Putfield` case. It gives a feeling on what steps must be taken also in the other cases. The instructions `Invoke`, `Return` as well as `Throw` modify the frame stack and also require a lot of hand tuned proof steps. The remaining instructions are straightforward and all their induction cases can be handled automatically. □

6.6.3 Instantiating the Locales

Unlike the requirements in the last section verifying the remaining locale requirements is straightforward. Since the lemmas look exactly as the requirements, we omit them here. After all requirements have been shown, we can instantiate our locales. Each locale comes with a predicate over all its parameters. This predicate is a conjunction of all the locale's requirements including those of its parent locales. The following theorems show with which setting of parameters, we have been able to instantiate our framework with.

Theorem 6.1 The requirements of locale *correctVCG* hold for these parameters:
correctVCG initS effS $\llcorner T \lrcorner$ $\llcorner F \lrcorner$ $\bigsqcap$ $\Rrightarrow$ $\models \vdash$ *ipc anF succsF wf initF wpF*

Theorem 6.2 The requirements of locale *completeVCG* hold for these parameters:
completeVCG effS $\llcorner T \lrcorner$ $\llcorner F \lrcorner$ $\bigsqcap$ $\Rrightarrow$ $\models$ *domC ipc anF succsTyF wf initF wpF*

Theorem 6.3 The requirements of locale *invariantVCG* hold for these parameters:
invariantVCG initS effS $\llcorner T \lrcorner$ $\llcorner F \lrcorner$ $\bigsqcap$ $\Rrightarrow$ $\models$ *safeF domC ipc anF succsF wf initF wpF*

Theorem 6.4 The requirements of locale *Expressiveness* hold for these parameters:
Expressiveness effS $\llcorner T \lrcorner$ $\llcorner F \lrcorner$ $\bigsqcap$ $\Rrightarrow$ $\models$ *domC ipc anF succsF wf initF wpF specF*

In Theorem 6.4 the paramter *specF* refers to a function, which views *initF* Π as precondition and all annotations at final positions as postconditions.

specF :: *jbc-prog* ⇒ *pos* ⇒ *expr option*
specF Π *p* = *if p* = *ipc* Π *then* ⌊*initF* Π⌋ *else if p* ∈ *finals* Π *then* ⌊*aF* Π *p*⌋ *else None*

6.7 Correctness and Completeness Theorems

By now we have defined all the parameter functions our abstract framework expects for Jinja. We have also proven that they satisfy the requirements stated in the framework.

This enables us to instantiate the theorems proven abstractly within the framework with our concrete VCG, as defined in §6.4.

6.7.1 Correctness

Most important is the following theorem, which guarantees correctness of our Jinja verification condition generator *vcg*:

Theorem 6.5 Jinja programs with provable verification condition are safe and have correct annotations.

$$\frac{wf\ \Pi \qquad \Pi \vdash vcg\ \Pi}{isSafe\ \Pi \wedge correctAn\ \Pi}$$

Theorem 6.5 is a combination of theorems 2.1 and 2.2. Now, *wf*, *vcg*, $\vdash$ and *isSafe*, which internally uses *initS*, *effS*, $\models$ and *safeF*, are the concrete functions we defined for Jinja. We have proven that the generic VCG instrumented by the control flow function and abstract semantics defined in §5 is correct. Theorem 6.5, just as all the other theorems and lemmas in this chapter, is proven outside any locale and does not depend on any further assumptions. Although not shown here, we have verified exactly the same correctness result for the other VCGs *vcgFrNr*, *vcgExTys* and *vcgTy*. With the upgrade theorem 2.7 and the lemmas on the system invariants 6.2, 6.3 and 6.4 we can derive this from theorem 6.1.

6.7.2 Invariance

In a similar fashion we can carry over a theorem that guarantees verification conditions to be invariants. That means, the formula *vcg* Π holds for all reachable states.

Theorem 6.6 Safe and correctly annotated Jinja programs, have an invariant verification condition.

$$\frac{wf\ \Pi \qquad isSafe\ \Pi \qquad correctAn\ \Pi}{\forall\, s{\in}Reachables\ \Pi.\ \Pi,s \models\ vcg\ \Pi}$$

This property can be seen as a weak variant of completeness. It is not equivalent to semantical completeness, which would demand the verification condition to be a tautology. Nevertheless, it indicates that *vcg* does not reject programs blindly. A function yielding $\lfloor F \rfloor$ for every input would also be safe in the above sense, but clearly not yield invariants. We also have a framework for semantical completeness, but we cannot instantiate it for *vcg*. The reason is that one of its requirements demands the successor function to guarantee progress. This is not the case for *succsF*, whose branch conditions provide valuable restrictions, but do not prevent *effS* from getting stuck. For example in case of `IAdd` the transition relation *effS* gets stuck if there are not at least two elements on the operand stack. The branch condition *succsF* yields for `IAdd` is $\lfloor T \rfloor$, which would also hold for such malformed states.

6.7.3 Completeness

With *succsTyF*, which augments the branch conditions of *succsF* with various system invariants, we cure this defect. In *vcgTy* we instantiate our generic VCG with *succsTyF* instead of *succsF*. Since *succsTyF* does guarantee progress, we can show semantical completeness for this VCG.

Theorem 6.7 Strongly annotated Jinja programs have a tautologous verification condition.

$$\frac{wf\ \Pi \qquad strongAn\ \Pi}{\forall s.\ \Pi,s \models\ vcgTy\ \Pi}$$

When annotations are strong enough is defined in §2.8. Roughly speaking, this means if we have an annotation A at p, then we can start the program at p with any state satisfying A and have that the program runs safely and satisfies all further annotations coming along. The important gain of theorem 6.7 towards theorem 6.6 is in the conclusion. Now, the obtained formula is a tautology and does not just hold for all reachable states. Provided the safety logic is complete, this means we have a provable verification condition. Unfortunately this is not the case for our safety logic. It embodies natural numbers and is thus naturally incomplete. However, if we fail to prove a certain program it is because of the (unavoidable) incompleteness of first order arithmetics and not because our VCG is overly restrictive.

6.8 Conclusion

The safety policy we defined in this section is just one example. One could replace it with any other safety policy expressible in our assertion language without having to modify any proof.
The requirement proofs mainly concentrate on the relation between the control flow function and abstract semantics towards the concrete semantics. Both have to be static approximations of the dynamic behaviour. The proofs about the weakest precondition operator turned out to be the largest and most difficult part of our work. In total it comprises 9kloc of Isar proof scripts. One reason why these proofs are so large is because we make nested case distinctions on two large datatypes (expressions and instructions). In [96] we also had this situation, but followed the standard approach of deriving substitution lemmas [99]. That means we have lemmas that allow us to reduce the substitution by making modifications on the state.

$evalE\ \Pi\ s\ (substE\ ((x,y) \cdot mp)\ Q) = eval\ \Pi\ (eff\ (x,y)\ s)\ (substE\ mp\ Q)$

The modification function *eff* (x,y) compensates on the state s the difference of evaluating y instead of x in Q. One keeps applying such lemmas until the substitution map becomes empty, i.e. $[]$, and the modifications of *eff* accumulate to the full state transition of *effS*. In case of Jinja this standard approach did not work well, because Jinja has instructions with so many different effects. We would need almost as many substitution lemmas as instructions. For this reason we skip these lemmas and evaluate substitutions directly for each instruction.

Also *succsF* requires non-trivial proofs. Proving the correctness property on *succsF* takes 2.5kloc of Isar proofs and the progress property 3.5kloc. In both cases the main difficulty is to prove lemmas that show the correlation between *find-handler* (part of *effS*) and *succsXpt* (part of *succsF*). The remaining proofs are easy (less than 1kloc). In the final conclusion of this thesis we show a table with all the sizes.

In this section we also showed how one can achieve modularity for each method without using specific call and return rules. Our idea is to pack call context specific information into branch conditions. Note that methods and their modular specification also have a downside. Now, we not only have to verify safety, but also functional correctness. Detecting modular method specifications automatically is difficult in practice, and so manual annotations from programmers are likely to be required here.

7 Generating Annotations and Proofs

By now we can produce verification conditions against arithmetic overflow for annotated Jinja programs. Two important questions remain. First, where do the annotations come from? Second, how can we prove verification conditions? Both questions are the topic of this chapter.

Let us first consider annotations. Clearly they could be inserted manually by the programmer. The fact that we are working on the bytecode level is no real obstacle. The data abstractions of Java and its bytecode are almost identical. Hence, one could write a tool that translates assertions in the source code, e.g. in JML [58], into our assertion language for the bytecode. The bigger problem is, that programmers are not (yet?) used to writing annotations. Having an automatic method to infer those, would be much more practical. For some safety policies, such as type- or control flow safety, inferring annotations is a standard technique by now. It happens silently inside compilers. For less abstract policies, such as absence of overflow, this is not standard. For these purposes program analysers can be used.

7.1 Program Analysis

At the moment most program analysers [41, 21, 60] only support bug detection, but the information they extract is also useful for program verification [59]. Since we have chosen overflow safety, we are interested in finding arithmetic properties. Interval analysis, such as the one implemented for Jinja bytecode [28] by Amine Chaieb, is very helpful in that respect. In addition, the Jinja bytecode verifier modelled and verified in Isabelle/HOL by Gerwin Klein [55, 54] is a valuable provider of annotations.

7.1.1 Bytecode Verifier

In *inv-Ty* from §6.3, we use *prog-kil* to infer primitive Jinja types using Kildall's dataflow algorithm. The Jinja article[55] reveals the details of this construction. The result of *prog-kil* is a mapping from program positions to so-called state types. These are either of the form *Err*, *OK None* or *OK* $\lfloor(st,\ lt)\rfloor$. If the state type *Err* occurs somewhere,

instruction	stack	registers
`Load` *0*	(	[], [*Class B*, *Integer*])
`Store` *1*	(	[*Class A*], [*Class B*, *Err*])
`Load` *0*	(	[], [*Class B*, *Class A*])
`Getfield` *F A*	(	[*Class B*], [*Class B*, *Class A*])
`Goto` -3	(	[*Class A*], [*Class B*, *Class A*])

Figure 7.1: Example of a well typed program.

there is a type error and the program is rejected by the type checker. If a position has state type *OK None*, it has not yet been visited by the algorithm. Initially all positions are marked with *OK None*. State types of the form *OK* $\lfloor(st, lt)\rfloor$ abstract the topmost frame with the types of its operand stack *st* and local variables *lt*. Both, *st* and *lt* are lists of Jinja types *ty* with an additional type *Err* representing uninitialised or erroneous values. For example, *OK* ([*Integer*],[*Class B*,*Err*]) represents a state type, which states that an integer is lying on the stack, while register *0* contains a reference to a *B* object and register *1* an erroneous value. An exemplary program *P* together with its state types Φ is shown in Fig. 7.1. For the instructions on the left hand side *prog-kill P* infers the state types on the right. The type checker *wf-jvm-prog-phi* Φ *P* (see [55]) accepts this program, because all instructions receive arguments with proper type. Note that, although the first instruction pushes a *B* reference onto the stack, we have *A*, the super class of *B*, on the operand stack in the succeeding state type. This is because the succeeding position is a join point, where *prog-kil* merges types to their least common supertype. Register 1 at that position becomes typed with *Err*, the least common supertype of *Integer* and *Class A*.

Using *annotate-types* and *convert-pt* as auxiliary functions, we can transform these state types into an assertion. This assertion is a conjunction of *Ty ex tp* expressions, where *ex* is either of the form *Rg k* or *St k* and *tp* the corresponding type. The example given above would translate to the assertion *Ty* (*St 0*) *Integer* $\wedge$ *Ty* (*Rg 0*) (*Class B*). Error types are simply dropped during the translation. Using the type soundness proof of [55] we can derive lemma 6.4, stating that *inv-Ty* is a system invariant.

7.1.2 Interval Analysis

To verify arithmetic overflow it is very helpful to have upper and lower bounds for integer variables available. Interval analysis, such as the one implemented for Jinja bytecode

instruction	stack	registers	source
12 `Load` *2*	([],	[[−*1*,*4*], [−*2*,*6*]])	
13 `Load` *1*	([*Rg 2*],	[[−*1*,*4*], [−*2*,*6*]])	
14 `IfIntG` +7	([*Rg 1*, *Rg 2*],	[[−*1*,*4*], [−*2*,*6*]])	`int m(int x, int y) {`
15 `Load` *1*	([],	[[−*1*,*4*], [−*2*,*4*]])	`if (-1 <= x & x <= 4`
16 `Load` *2*	([*Rg 1*],	[[−*1*,*4*], [−*2*,*4*]])	`&-2 <= y & y <= 6)`
17 `IMul`	([*Rg 2*,*Rg 1*],	[[−*1*,*4*], [−*2*,*4*]])	`{ if (y<=x) {`
18 `Store` *2*	([*Rg 1* ⌞*⌟ *Rg 2*],	[[−*1*,*4*], [−*2*,*4*]])	`y=x*y;`
19 `Push` *0*	([],	[[−*1*,*4*], [−*8*,*16*]])	`x=0; }`
20 `Store` *1*	([[*0*,*0*]],	[[−*1*,*4*], [−*8*,*16*]])	`return y;`
21 `Load` *2*	([],	[[−*1*,*4*], [−*8*,*16*]])	`}`
22 `Return`	([*Rg 2*],	[[−*1*,*4*], [−*8*,*16*]])	`}`

Figure 7.2: Arithmetic program

[95, 28], provides such information. The basic idea is that integer values are abstracted with intervals. For example the interval $[0,3]$ abstracts the concrete value of a variable that may range from *0* to *3*. The state types consist of a list of intervals for the registers and a list of expressions for the operand stack. Using expressions for the stack instead of intervals allows to exploit branch conditions for sharpening interval bounds.

In Fig. 7.2 we have a small Java program manipulating variables x and y, which are represented in registers *Rg 1* and *Rg 2* in the bytecode. We only show the bytecode snippet for the inner conditional, which we enter with the intervals $[-1,4]$ for *Rg 1* and $[-2,6]$ for *Rg 2*. Note how storing expressions on the stack pays off for the conditional jump from position 14 to 15. The analyser knows the condition $Rg\ 2 \leqslant Rg\ 1$ and that the arguments *Rg 1* and *Rg 2* are lying on the stack. Combining these facts allows it to restrict *Rg 2* to $[-2,4]$ at position 15. If we only had intervals on the stack this restriction would not be possible. An easy way to use the intervals shown in Fig. 7.2 for our purposes is to translate them into annotations. For example for position 12, we could generate the following formula:

$$St\ 0 = none \wedge \lfloor Intg\ -1 \rfloor \leqslant Rg\ 1 \wedge Rg\ 1 \leqslant \lfloor Intg\ 4 \rfloor \wedge \lfloor Intg\ -2 \rfloor \leqslant Rg\ 2 \wedge Rg\ 2 \leqslant \lfloor Intg\ 6 \rfloor$$

Since annotations need to be verified in the verification conditions, we do not have to trust the analyser. If the annotations are incorrect, we end up with an unprovable verification condition.

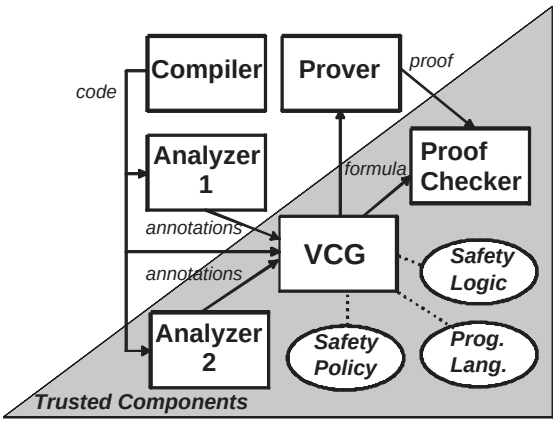


Figure 7.3: PCC system architecture

7.2 Integrating Trusted and Untrusted Analysis Results

In the previous section, we have seen two examples of program analysis, bytecode verification and interval analysis. Both yield formulas in our assertion logic, but there is a difference. The bytecode verifier has been formalised in Isabelle/HOL and proven to be correct. Theorem 6.4 from §6.3 expresses this correctness, the inferred types hold at runtime. The interval analysis on the other side is not proven to be correct [55]. Hence, we cannot trust its results. Now, the question arises how we can integrate trusted and untrusted analysis results in a meaningful way [95]. In Fig. 7.3 we illustrate the situation in general. We have a trusted analyser and an untrusted one. Our goal is to instantiate the VCG such that it produces proof obligations that require us to show the validity of untrusted facts and allow us to exploit the trusted facts.

Assume that for a given program P the first analyser gives the annotations $aF_1\ P$ and the second one $aF_2\ P$. Both are finite maps from positions to assertions expressing the facts each analyser has found. A simple way to integrate both results is to conjoin the maps. We take $An_{12} = conjAn\ (aF_1\ P)\ (aF_1\ P)$ and construct the annotated program $\Pi_{12} = (P,An_{12})$. The function *conjAn* conjuncts the annotations of two maps. For example, if the first map yields A_1 for a position p and the second one A_2, i.e. $aF_i\ P\ p = \lfloor A_i \rfloor$, then An_{12} maps p to $A_1 \mathbin{\triangle} A_2$. If one map has a mapping where the other one does not, the missing annotation is replaced with $\lfloor T \rfloor$. Only if both maps do not have an annotation for a position p, An_{12} does not have one either, i.e. $An_{12}\ p = None$.

To verify safety of Π_{12} and the correctness of its annotations, we can construct and

prove the verification condition $vcg\ \Pi_{12}$. Assume position p has a successor p' with branch condition B, i.e. $(p',B) \in set\ (succsF\ \Pi\ p)$. Also assume that the analysers give us the annotations A_1' and A_2' for the successor p', i.e. $aF_i\ P\ p' = \lfloor A_i' \rfloor$. Then the verification condition contains the following proof obligation:

$(safeF\ \Pi\ p \between (A_1 \between A_2) \between B) \Rightarrow\!\!\!\!\lrcorner\ wpF\ \Pi\ p\ p'\ (safeF\ \Pi\ p' \between (A_1' \between A_2'))$

The problem with this proof obligation is that annotations appear on the right hand side of the implication $\Rightarrow\!\!\!\!\lrcorner$. This means, we have to verify their correctness, when we prove the verification condition. This is acceptable if neither analyser can be trusted. Now, assume the second analyser has been verified and all the annotations it gives are known to be correct. Then, we are in an unhappy situation, because we have to verify A_2' again. A way around this problem is to pack trusted annotations into branch conditions. These only appear on the left hand side of implications $\Rightarrow\!\!\!\!\lrcorner$. This can be accomplished by instantiating our verification condition generator with a different successor function $succsF'$. Using the upgrade function *upg* from §2.10 we can add the formulas from the trusted analyser to the branch conditions the original successor function $succsF$ yields. Then we instantiate vcg', a verification condition generator that is identical to *vcg* from §6.4, except that it uses the upgraded successor function $succsF'$.

$vcg' = vcG\ \ldots\ initF\ (upg\ succsF\ (\lambda\ (P,An).\ aF_2\ P))\ wpF\ sF\ aF.$

Now, we take the annotated program $\Pi_1 = (P,aF_1\ P)$, which only carries the annotations from the first analyser, and verify it with the new VCG. The new verification condition $vcg'\ \Pi_1$, gives us the following proof obligation for position p:

$(safeF\ \Pi\ p \between A_1 \between (A_2 \between B)) \Rightarrow\!\!\!\!\lrcorner\ wpF\ \Pi\ p\ p'\ (safeF\ \Pi\ p' \between A_1)$

The goal A_2' now has disappeared, but all the facts on the left hand side remain. This means we arrived at a verification condition that is easier to prove. The price we have to pay for this manipulation is that we have instantiated a new VCG and now have to prove again that it meets all the requirements. Here the instantiation theorem 2.7 from §2.10 helps. Provided we already have proven the requirements for *vcg*, we now only have to show that the formulas we upgrade the successor function with are system invariants. Since we said that the second analyser can be trusted, this follows from its correctness proof. If we associate aF_2 with our trusted bytecode verifier, this is exactly what theorem 6.4 in §6.3 guarantees. In §6.4 we follow this principle when we instantiate *vcgTy*. Since we only upgrade system invariants, we can justify the correctness of *vcgTy* from the correctness of *vcg*. Although *vcgTy* produces larger verification conditions than *vcg*, they are easier to prove as more facts are available. Using simple optimisation techniques as the ones shown in the next section, help to get rid of redundant facts, before starting the safety proof.

7.3 Optimising Verification Conditions

Making the VCG generic reduces its complexity and simplifies proofs of correctness and completeness. On the other hand, generic verification conditions tend to be very verbose. One reason is that our VCG does not abstract the control flow and produces many proof obligations for programs with complex control flow. In particular, exceptions lead to a lot of paths. Many of those turn out to produce the same or very similar verification conditions. However, there are ways to tackle these inefficiencies. One is to instantiate sophisticated parameter functions. For example *wpF* and *succsF* could analyse the safety policy and filter out irrelevant parts of the formulas they produce. This would keep the resulting verification conditions small from the start, but also make the definitions of these functions more complex and thus proving the requirements harder. Another way is to optimise the resulting verification condition in a post-processing step. Then instead of proving the real verification condition *vc* a code producer would only prove the reduced version *opt vc*. However, there are two things one must keep in mind. First, the optimiser *opt* :: *expr* ⇒ *expr* must preserve the validity of formulas, i.e. $\Pi,s \models vc = (\Pi,s \models opt\ vc)$. Otherwise correctness and completeness of the VCG become void. Second, *opt* must be very efficient, as it has to run on the consumers side too.

The good news is that there are many simple and efficient optimisations. In experiments we found that in case of Jinja such optimisations greatly impact the size of the resulting verification conditions and also of the resulting proofs. Factors between 4 and 10 are not seldom. We implemented three kinds of optimising functions. Their definitions are straightforward, hence we only describe them informally here. Our first optimiser *fA*:: *expr* ⇒ *expr* flattens conjunctions. It searches a given expression for nested conjunctions and transforms these into flat conjunctions. Here is an example:

fA (⋀ [*Rg 0* ⟝ *Rg 0*, ⋀ [*Rg 1*,*Rg 1*], (⋀ [⋀ [*Rg 2*,*Rg 3*],*Rg 2*]) ⟹ *Rg 3*])
= ⋀ [*Rg 0* ⟝ *Rg 0*,*Rg 1*,*Rg 1*,(⋀ [*Rg 2*,*Rg 3*,*Rg 2*]) ⟹ *Rg 3*]

Flattening conjunctions is quite helpful for our next optimisation *rd*::*expr* ⇒ *expr*, which removes duplicates. Not only does it remove duplicates in conjunctions, but also it removes subformulas on the right hand side of implications that also occur on the left hand side. Applied to the result from above, *rd* would reduce the implied *Rg 3* with ⟙ and eliminate redundant occurrences of *Rg 2* and *Rg 1*.

rd (⋀ [*Rg 0* ⟝ *Rg 0*,*Rg 1*,*Rg 1*,(⋀ [*Rg 2*,*Rg 3*,*Rg 2*]) ⟹ *Rg 3*])
= ⋀ [*Rg 0* ⟝ *Rg 0*,*Rg 1*,(⋀ [*Rg 2*,*Rg 3*]) ⟹ ⟙]

The implication above is an example of a constant subexpression. Its evaluation yields the same for any state. Our next optimiser *fc*:: *expr* ⇒ *expr* detects such constant subexpressions and folds them to their evaluation result. Here is what *fc* does to the result above.

$fc\ (\bigwedge [Rg\ 0 = Rg\ 0, Rg\ 1, (\bigwedge [Rg\ 2, Rg\ 3]) \Rightarrow T])$
$= \bigwedge [Rg\ 1]$

Note that *fc* is not complete, it only discovers simple cases of constant subexpressions. It reduces conjunctions, implications involving T or F, biased conditionals or syntactic equalities. It also optimises relations or numerical compositions of constant subexpressions, e.g. $Intg\ 1 + Intg\ 2 = Intg\ 3$. However, it would not discover that $Rg\ 0 \leq (Rg\ 0 + Intg\ 1)$ is always true. A complete *fc* would reduce every tautology to the constant T. This would amount to a decision procedure for first order arithmetics, which is impossible due to Gödel's incompleteness theorem.

For reasons of convenience we bundle our optimising functions into one optimiser.

$opt :: expr \Rightarrow expr.$
$opt = fc \circ rd \circ fA \circ fc$

Using similar theorems for the functions *fA*, *rd* and *fc*, we have proven the following theorem, which justifies optimising formulas prior to their proof.

Theorem 7.1 The optimiser *opt* preserves the semantics of expressions.
$evalE\ \Pi\ s\ (opt\ ex) = evalE\ \Pi\ s\ ex$

The complexity of *opt* is quadratic to the size of the input formula. This is because some functions, e.g. *rd*, need to look up expressions in lists of subexpressions. The quadratic time is acceptable, because our verification conditions typically are conjunctions of many small subgoals that can be optimised independently. In fact we could also justify the use of less efficient optimisers, because the main bottleneck lies in producing and writing down proofs. For example decision procedures for Presburger arithmetics can produce proofs that are double exponentially bigger than the proven formula [45].

Although the optimisations *opt* performs are rather simple, their effect on verification conditions is enormous. For example optimising the verification condition shown in Fig. 6.2 leads to the formula shown in Fig. 7.4, which has ten times fewer nodes. Note that all redundant branch conditions have been removed and the conditionals in the postcondition have been simplified. Like Fig. 6.2 we also omit the parts for the exceptional behaviour of *up* with $\_\_$. These parts are also reduced drastically, some even collapse to T.

```
opt-vc-Cnt-up-0 = ⋀ [Ty (Rg 0) (Class Cnt), Ty (Rg 1) Integer,
1  Rg 1 ≤ ₌Intg 2147483647₌, Rg 0 ≠ (Call (St 1)),
2  Rg 1 ≠ (Call (St 0)), Gf c Cnt (Rg 0) ≠ (Call (Gf c Cnt (St 1))),
3  Pos (Cnt, up, 0), ₌Intg 1₌ ≤ FrNr]
4
5  ⇒
6
7  (⋀ [---, ⋀ [¬ (Rg 1 ≤ ₌Intg 0₌)]
8
9    ⇒
10
11 (⋀ [(₌Intg 2147483647₌ − Rg 1) ≤ ₌Intg 2147483647₌,
12  ⋀ [---, ¬ (Rg 0 ≠ ₌Null₌)]
13
14      ⇒
15
16 (⋀ [(₌Intg 2147483647₌ − Rg 1) ≥ (Gf c Cnt (Rg 0))]
17
18        ⇒
19
20 (⋀ [((Gf c Cnt (Rg 0)) + Rg 1) ≤ ₌Intg 2147483647₌,
21 ((Gf c Cnt (Rg 0)) + Rg 1) ≠ ((Call (St 0)) + (Call (Gf c Cnt (St 1))))])])])
```

Figure 7.4: Optimised Verification Condition: Body `Cnt.up`

7.4 Generating Proofs

To construct proofs for verification conditions we use a theorem prover. Although other first order provers with proof objects and sufficient support for arithmetics could be employed, we also use Isabelle/HOL for this purpose. This enables us to define the provability judgement semantically and spares us from verifying a calculus of inference rules. An alternative we also investigated is to let the program analysis (interval analysis) produce a proof for its result.

7.4.1 Proof Construction with Isabelle

As we have defined our provability judgement semantically, we can prove $\Pi \vdash f$ by proving $\forall\, s.\ \Pi, s \models f$. The latter amounts to evaluating the formula f on an unknown state s. That is we have to prove:

$\forall\ s.\ evalE\ \Pi\ s\ f = \lfloor Bool\ True \rfloor$

A proof for such a goal typically involves two steps. First, one uses the simplifier to

push *evalE* as far inside f as possible. The result is a HOL formula that connects various occurrences of *evalE* Π *s ex*, where *ex* is an atomic subexpression of f, with Isabelle/HOL's logical and arithmetic operators. That is $\bigsqcup$ becomes $+$, $\bigsqcap$ becomes an iteration of $\wedge$ and so on. Then the second step of the proof is to show that the resulting HOL formula holds. Here the huge library of Isabelle theories can be used as well as various decision procedures. In case of a safety policy against arithmetic overflow one often ends up with proving arithmetic subgoals. Here, Isabelle's decision procedure for Presburger arithmetic is useful. It automatically handles all linear arithmetic subgoals. In our experiments we found that in most cases the `clarsimp` procedure proves goals automatically. It performs rewriting, instantiates quantifiers and finally calls the arithmetic decision procedures. However, to get it running we had to introduce a lot of rewriting rules for the *evalE* function. Some of these are shown in Fig. 4.4.

7.4.2 Proof Producing Program Analysis

Instead of standard proof procedures, a different approach is to let the program analyser produce a proof of its result. After the analyser reaches a post-fixpoint, it should return a proof that the found annotations are correct. Intuitively, since the analyser infers these invariants, their correctness proofs will rely on a small set of theorems, which express how the analyser manipulates domain elements. Independently from [86] this technique is proposed in [28] and has been implemented in [39]. For every edge (p,q) in the control graph of a Jinja program it creates a theorem of the form *form* $P \longrightarrow$ *form* (*WP c* Q), where P and Q are domain elements inferred for p and q, and *form* is a function turning these into HOL formulas. The *WP* operator performs the same transformations as *wpF*, but works directly on domain elements (lists of intervals and stack expressions). For example take $p = (Start,m,17)$ and $q = (Start,m,18)$ and consider the program in Fig. 7.2. In this case, we obtain the following formulas:

form $P = -1 \leq r_1 \wedge r_1 \leq 4 \wedge -2 \leq r_2 \wedge r_2 \leq 4 \wedge -2 \leq s_0 \wedge s_0 \leq 4 \wedge -1 \leq s_1 \wedge s_1 \leq 4$

form $Q = -1 \leq r_1 \wedge r_1 \leq 4 \wedge -2 \leq r_2 \wedge r_2 \leq 4 \wedge -8 \leq s_0 \wedge s_0 \leq 16$

Note that *form* introduces integer variables r_1, r_2, s_0 and s_1 and resolves stack expressions into their intervals. The proof producing analyser emits the following lemma for the edge from p to q and also supplies an Isabelle/HOL proof for it.

lemma *p-q*:
$-1 \leq r_1 \wedge r_1 \leq 4 \wedge -2 \leq r_2 \wedge r_2 \leq 4 \wedge -2 \leq s_0 \wedge s_0 \leq 4 \wedge -2 \leq s_1 \wedge s_1 \leq 4$
$\longrightarrow$
$-1 \leq r_1 \wedge r_1 \leq 4 \wedge -2 \leq r_2 \wedge r_2 \leq 4 \wedge -8 \leq s_0 * s_1 \wedge s_0 * s_1 \leq 16$

The lemma $p{-}q$ involves multiplication and can thus not be proven automatically by

```
constdefs vc-Start-m-18- :: expr
vc-Start-m-18- =
  ⌊⋀⌋ [⌊Intg −1⌋ ⌊≤⌋ Rg 1, Rg 1 ⌊≤⌋ ⌊Intg 4⌋,
  ⌊Intg −2⌋ ⌊≤⌋ Rg 2, Rg 2 ⌊≤⌋ ⌊Intg 4⌋,
  St 0 ⌊=⌋ (Rg 1 ⌊*⌋ Rg 2),
  Pos (Start, m, 18)]

⌊⇒⌋

(⌊⋀⌋ [⌊Intg −8⌋ ⌊≤⌋ St 0, St 0 ⌊≤⌋ ⌊Intg 16⌋])
```

Figure 7.5: Verification Condition: (Start,m,18) to (Start,m,19)

Isabelle/HOLs decision procedures for arithmetics. We have a tactic for bounded arithmetic, which would solve the goal, but it is not included in the standard proof procedures such as `clarsimp`. Hence, having such lemmas proven by the analyser can be quite helpful. Consider position $r = (Start,m,19)$ in Fig. 7.2. When we translate the analysis results into our assertion format, we obtain annotation Ar.

```
constdefs Ar::expr
Ar = ⌊⋀⌋ [⌊Intg −1⌋ ⌊≤⌋ Rg 1, Rg 1 ⌊≤⌋ ⌊Intg 4⌋,
          ⌊Intg −8⌋ ⌊≤⌋ Rg 2, Rg 2 ⌊≤⌋ ⌊Intg 16⌋ ]
```

Analogously Q becomes translated to Aq.

```
constdefs Aq::expr
Aq = ⌊⋀⌋ [⌊Intg −1⌋ ⌊≤⌋ Rg 1, Rg 1 ⌊≤⌋ ⌊Intg 4⌋,
          ⌊Intg −2⌋ ⌊≤⌋ Rg 2, Rg 2 ⌊≤⌋ ⌊Intg 4⌋,
          St 0 ⌊=⌋ (Rg 1 ⌊*⌋ Rg 2)]
```

When we compute the optimised verification condition for this example we get *vc-Start-m-18-* as proof obligation for the transition from q to r. This formula, which we show in Fig. 7.5 is the result of $⌊⋀⌋\ [Aq,⌊T⌋,⌊T⌋] \Rightarrow wpF\ \Pi\ q\ r\ ⌊⋀⌋\ [Ar, ⌊T⌋]$. The constants $⌊T⌋$ are safety formulas or branch condition, they are optimised away in Fig. 7.5.

In Fig. 7.6 we show an Isar proof script for this verification condition. As one can see `clarsimp` cannot prove the condition *vc-Start-m-18-* automatically. It terminates with a subgoal, where one has to bound the multiplication $n2 * n2a$, where $n2$ has been introduced for the integer in register 1 and $n2a$ for the one in register 2. A similar problem occurs for the verification condition *vc-Start-m-0-* which covers the whole path

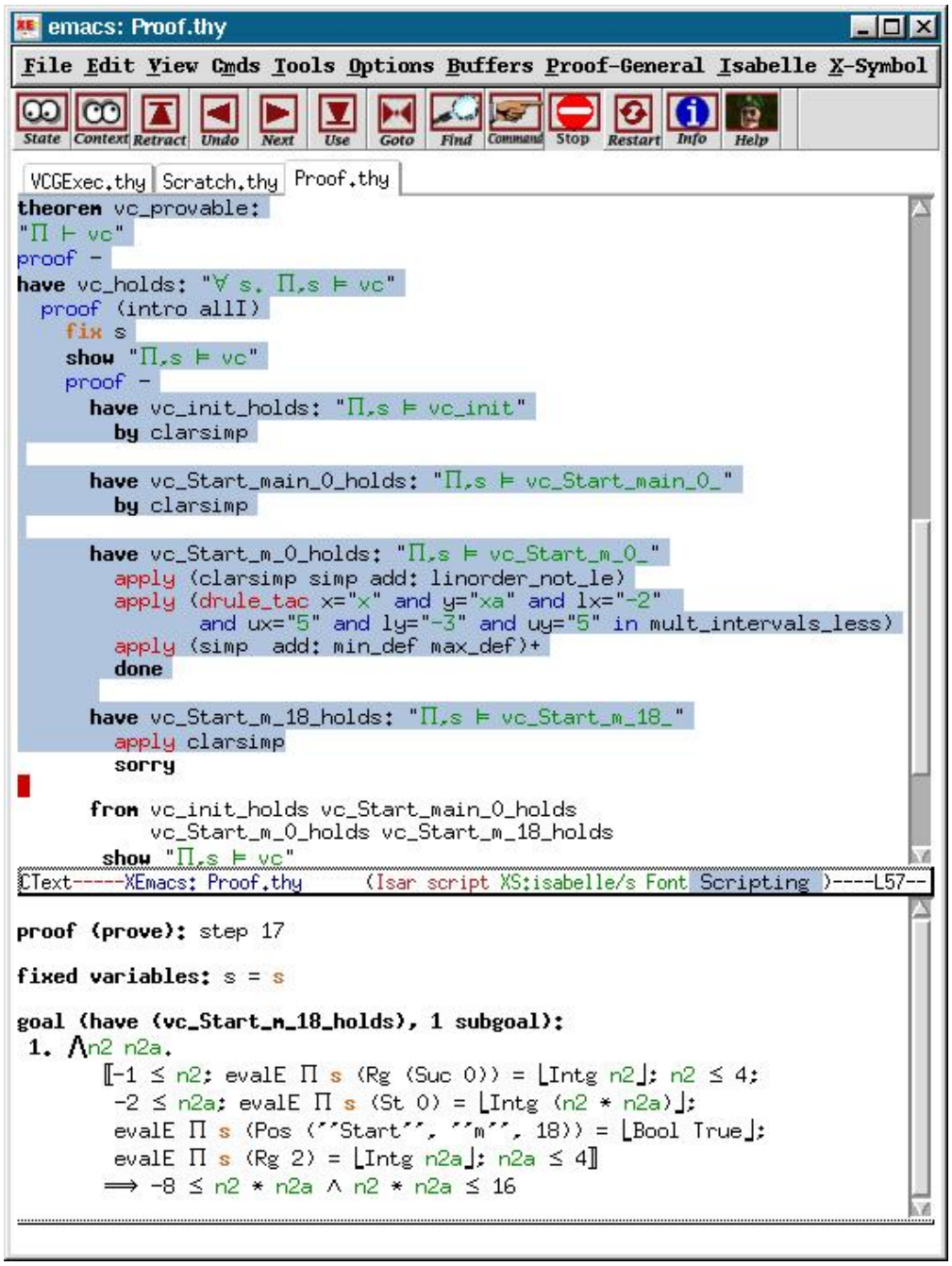


Figure 7.6: Proving the verification condition

from (*Start*,*m*,*0*) to (*Start*,*m*,*18*). There the safety formula for the `IMul` instruction requires to show that the result is within the representable range. To solve this subgoal we manually used a rule for interval multiplication.

lemma *mult-intervals-less*:
$lx < x \wedge x < ux \wedge ly < y \wedge y < uy \wedge l = [lx*ly, lx*uy, ux*ly, ux*uy] \longrightarrow$
$(foldl\ min\ (hd\ l)\ (tl\ l) < x * y \wedge x * y < foldl\ max\ (hd\ l)\ (tl\ l))$

We could use the same rule for *vc-Start-m-18-*, but as Fig. 7.6 shows one needs to give explicit bounds. Here the automatically generated theorems from the program analysis help. If we added lemma *p-q* from above to the simplification set we do not need the lemma *mult-intervals-less*. We could replace the three lines with `apply` we have in Fig. 7.6 for the goal `vc_Start_m_0_holds` with an automatic proof, namely with `by clarsimp`. The proofs obtained with proof producing program analysis are also a lot shorter. Arithmetical rewriting as we perform it for `vc_Start_m_0_holds` in Fig. 7.6 leads to very verbose proof objects. Having perfectly matching lemmas from the analyser available, avoids this. Note that all the lemmas and proofs from the analyser must be added to the proof for the verification condition, but this is just a technical issue.

7.5 Conclusion

This chapter showed how program analysis can be integrated into our PCC system. The best way is when one can trust the analyser. In this case the results can be placed into branch conditions. This makes them available as facts, but does not require to verify them. The only drawback with trusted analysers is that they must be installed and run on the consumer's side as well. This means they have to be efficient and a producer cannot adjust them for each individual program. Theoretically one could think about sending an individual analyser together with its correctness proof along with each program. However, our experience with the verified BCV [54] tells us that such correctness proofs are complex and big theories in HOL. In case an analyser cannot be trusted, which is the case for the interval analysis used here, one can use the results as annotations. The resulting verification condition can then be proved either interactively in Isabelle/HOL or by employing a proof producing program analyser [28, 39]. The latter case is of course preferable, but many analysers around these days do not provide this feature. In our case the interval analyser has been upgraded to produce proven lemmas for all edges in the control flow graphs. If we add these lemmas to the simplification set, the proof procedures of Isabelle/HOL can verify all annotations automatically.

8 Using the System

This chapter consists of two parts. First, we show how we can use Isabelle's code generator to turn our formalisation into a collection of runnable ML tools. Then we show on a small example how these tools should be applied by the code producer and consumer.

8.1 Generating Runnable ML Prototypes

Isabelle/HOL provides a code generator [17] for turning HOL specifications into runnable ML code. When instantiating our framework to Jinja, we carefully chose definitions that work well with this code generator. However, some definitions we inherit from Jinja [55] are not directly amenable for code generation.

For such situations, the code generator allows us to bridge the gap by providing alternative definitions via so called code lemmas. We had to do so for Jinja's functions looking up methods or fields in programs. In [55] these functions use existential quantifiers and Hilbert's choice operation *THE x. P x*, which returns a value satisfying a given predicate. We could make these functions executable, by turning them into inductive sets, for which the code generator can produce search algorithms. For example, consider the following definition:

$f\ x = THE\ y.\ \exists z.\ R\ x\ y\ z$

When the relation $R\ x\ y\ z$ enforces unique values for y for a given x, we can define an inductive set F with one introduction rule:

$R\ x\ y\ z \longrightarrow (x,y,z) \in F$

If R is already inductively defined, we can skip this step. Next, we turn the definition above into a conditional rewriting rule and declare it as an inductive code lemma.

lemma [code ind]: $(x,y,z) \in F \longrightarrow f\ x = y$

The code generator then generates code for f that internally searches the inductively defined set F for a solution, quite similar to a Prolog [23] interpreter. Apart from that, the code generator can also use alternative types. In our case, we replaced finite sets with

lists. We do not go further into the details of code generation here. Interested readers may have a look at our theory files `BCVExec.thy` and `VCGExec.thy` [4], which contain all the code lemmas, type conversions and the like. The generated system consists of three executable ML programs. All together they make up 4.3kloc of Standard ML [63].

(1) Bytecode Verifier.

(2) Wellformedness Checker.

(3) Verification Condition Generators and Optimiser.

Each of those is placed into a separate ML structure inheriting the common datatypes from other structures defining the Jinja syntax and the expression language. For the convenience, we have ML startup scripts for each component: `concon`, `jinwf` and `jinvcg`. These scripts not only invoke the generated ML modules, but also perform additional tasks. The tool `concon` turns the bytecode verifier's types into control constraints, `jinwf` not only answers with yes/no, but also gives detailed error descriptions and `jinvcg` decomposes the resulting verification condition and also produces a tailored Isar [93, 92] proof script for it. We discuss each of these tools in the following sections.

8.2 Tasks for Code Producers and Consumers.

This section illustrates the tasks code producers and consumers have to perform. Fig. 8.1 gives an overview. In total there are 13 steps to take, 1-9 for the producer and 10-13 for the consumer.

1. The producer writes a Java program and compiles it.
2. The producer uses `j2jin` to convert the classfile into Jinja bytecode in formats for ML and Isabelle/HOL.
3. Program analysers for types and intervals inspect the bytecode and produce annotations our assertion logic.
4. The producer loads these annotations into Isabelle/HOL, merges them automatically and maybe adds further annotations manually.
5. The wellformedness checker inspects the code and annotations and rejects them if it discovers defects.

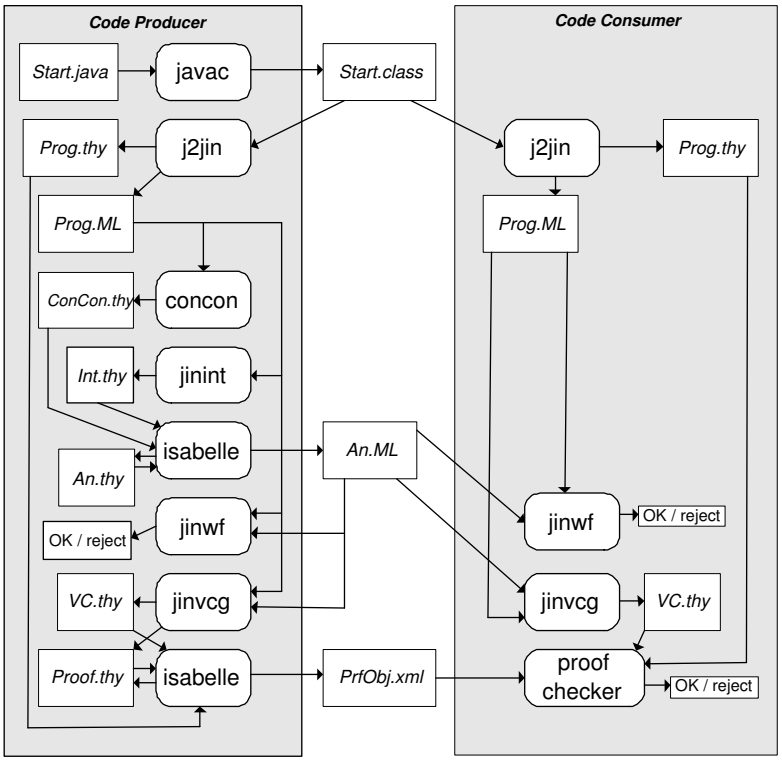


Figure 8.1: Jinja PCC - Workflow

6. The producer runs `jinvcg`, which turns program and annotations into a verification condition together with a proof script for it.
7. The producer loads the verification condition and the proof script into Isabelle/HOL and generates the proof.
8. When the proof is finished the producer exports it as an LF style proof object [16, 15] into a XML file.
9. The producer sends the classfiles, annotations and proof object to the consumer.
10. The consumer receives the classfiles and converts them to Jinja using `j2jin`.
11. The consumer runs the wellformedness checker on the converted code and the received annotations.
12. If the check is successful, the consumer runs the VCG to obtain the verification condition.

13. The consumer uses Isabelle's proof checker to find out whether the received proof object is a valid proof for the generated verification condition.

If the proof is valid, the program is guaranteed to run safely on the Jinja VM and is cleared for execution. Although our formalised Jinja VM could also be turned into an executable ML program, executing the original classfiles on a real Java VM gives much better performance. There is a risk that errors in `j2jin` or our formalisation of the Jinja VM, cause the code to behave differently. If the Jinja bytecode does not match the original classfile or the real Java VM [88] behaves differently than our formalised version, our safety guarantee cannot be transferred to the real world. However, here we are close to an absolute limit in safety concerns. At some point the gap between real and formal world is inevitable. The good news is that we have narrowed down the potential risks to a small number of trusted components. They can be checked once and for all by human inspection.

In the following sections, we illustrate all these steps with a small example. Instead of the counter application from Fig. 3.2, we now pick a program that is numerically more appealing and better suited for the interval analyser, which cannot handle method invocations and exceptions.

8.2.1 From Java to Jinja

Although programming directly in bytecode is possible, it is much more convenient to write code in Java or any other source language that can be compiled to Java bytecode. The example we discuss in this section is the Java program shown in Fig. 8.2. It contains a method *sum*, which adds the numbers from 0 to n.

$sum\ n = \sum_{i=0}^{n} i$

Instead of using Gausses famous formula, i.e. *sum n = n (n+1) / 2*, we rather keep the program more interesting by using a loop of additions. Our aim is to write this program in a robust form. It should check its input n and avoid running into arithmetic overflows. The constant *mn* denotes the largest input n, for which the result of *sum n* is still representable as a 32bit Java integer. If n exceeds this number the loop carrying out the additions causes an arithmetic overflow and we end up with a wrong result. Therefore, our method first checks whether n exceeds *mn* and immediately terminates with the error code *−1* if so. Otherwise it enters a loop that n times increments i and adds it to the result variable s. Both s and i grow with every iteration, but are always lower or equal to the result *sum n*, which our check on n already keeps within the representable range. Hence, in principle our initial check suffices to exclude arithmetic overflow. However, our code also checks the result variable s within the loop to be below

```
public class Start {

 static final int input = 100;
 static final int mn = 65535;
 static final int ms = 2147418112;

 public static void main(String args[]) {
  try { Start st=new Start();
        st.sum(input); }
  catch (Exception e) {}
 }

 public int sum(int n) {
  int i  = 0;
  int s  = 0;
  if (n <= mn) {
   while (i <= n) {
    if (s <= ms) { s=s+i;
                   i=i+1; }
    else {
     s=-2;
     //System.out.println ("Impossible!");
     break; }
   }
   //System.out.println(s);}
  else {
   //System.out.println("Input too high!");
   s=-1; }
  return s;
 }
}
```

Figure 8.2: Gauss Summation

the constant *ms*, which we have chosen relative to *mn* and *maxI*. That is, we have: *sum mn* = *ms* + *mn* = *2147483647* = *maxI*.

Note that *maxI* can be distinguished easily from *ms* by reading their digits from right to left. The reason for the cumbersome check `s <= ms`, which is always true, is that we want to use interval analysis to obtain proper annotations automatically. Interval analysis does not relate variables with each other and would widen the interval for s to $[0, \infty]$ otherwise. Since the relation between s and i is the non-linear Gaussian formula, we would need a very advanced program analyser to avoid the magic number *ms*.

After the code has been written, we use the java compiler to obtain the classfiles.

```
:) javac Start.java
```

We obtain `Start.class`, which we now have to convert to the Jinja format using `j2jin`.

```
:) j2jin -o Prog Start.class
```

This gives the following output:

```
File Prog.ML has been generated.
File Prog.thy has been generated.
```

The resulting files `Prog.ML` and `Prog.thy`, which we partly show in Fig. 8.3, contain Jinja representations of *Start.class* in formats for ML and Isabelle. In Fig. 8.4 we show the control flow of this code.

The ML file is processed further on by our tools, while the Isabelle file is needed for verification in Isabelle/HOL only. When `j2jin` finds an instruction Jinja does not directly support, it replaces it with a proper substitute. For example *System.out.writeln*, which we commented out to avoid clutter in the bytecode, would not be translated into `Invoke` *Printstream* ..., but result in a series of `Pop` instructions that just clear the arguments. Whenever such a substitution is made the ML file contains a comment with the real instruction. The generated theory file decomposes the entire program into various constant definitions. For each method, we get a separate constant listing all its instructions commented with their line of occurrence. To integrate these comments we use the $--$ operator, which simply drops its first argument: *linenr* $--$ i = i.

theory Prog **imports** VCGExec **begin**

. . .

constdefs m-Start-sum:: jvm-method mdecl
m-Start-sum = (sum, [Integer], Integer, 3, 3,
[0 −− `Push` (Intg 0),
1 −− `Store` 2,
2 −− `Push` (Intg 0),
3 −− `Store` 3,
4 −− `Load` 1,
5 −− `Push` (Intg 65535),
6 −− `IfIntG` 19,
7 −− `Load` 2,
8 −− `Load` 1,
9 −− `IfIntG` 18,
10 −− `Load` 3,
11 −− `Push` (Intg 2147418112),
12 −− `IfIntG` 10,
13 −− `Load` 3,
14 −− `Load` 2,
15 −− `IAdd`,
16 −− `Store` 3,
17 −− `Load` 2,
18 −− `Push` (Intg 1),
19 −− `IAdd`,
20 −− `Store` 2,
21 −− `Goto` −14,
22 −− `Push` (Intg −2),
23 −− `Store` 3,
24 −− `Goto` 3,
25 −− `Push` (Intg −1),
26 −− `Store` 3,
27 −− `Load` 3,
28 −− `Return`],
[])

constdefs StartC:: jvm-method cdecl
StartC = (Start, Object,
[(input, Integer),(mn, Integer),(ms, Integer)], [m-Start-main, m-Start-sum])

constdefs P:: jvm-prog
P = [ObjectC, ExceptionC, NullPointerC, ClassCastC, OutOfMemoryC, StartC]

end

Figure 8.3: Prog.thy

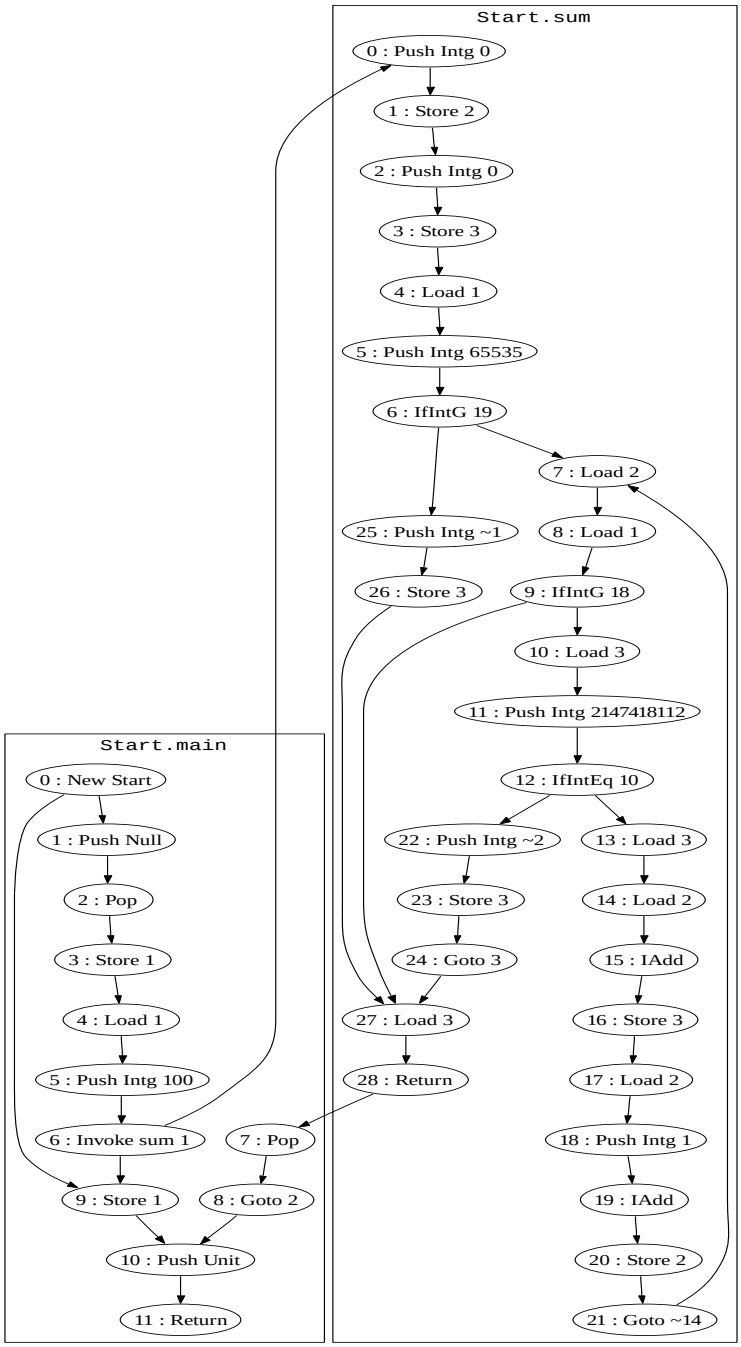


Figure 8.4: Gaussian Summation - Control Flow

theory ConCon **imports** JBC_SafetyLogic **begin**

constdefs concons :: pos ∼∼> expr
concons = [
((Start, main, 0), Ty (Rg 0) NT),

((Start, main, 6), $\bigsqcap$ [Ty (St 1) (Class Start), Ty (St 0) Integer]),

((Start, main, 7), $\lfloor\top\rfloor$), ((Start, main, 11), $\lfloor\top\rfloor$),

((Start, sum, 0), $\bigsqcap$ [Ty (Rg 0) (Class Start), Ty (Rg 1) Integer,
Rg 0 $\sqsupseteq$ (Call (St 1)), Rg 1 $\sqsupseteq$ (Call (St 0))]),

((Start, sum, 7), $\lfloor\top\rfloor$), ((Start, sum, 28), $\lfloor\top\rfloor$)]

end

Figure 8.5: ConCon.thy

8.2.2 Annotating the Code

Now, after we have the Jinja bytecode the next step is to annotate it. First, we generate so called control constraints, which are annotations our VCG expects.

```
:) concon -o ConCon Prog.ML
```

If the program is welltyped, this gives the following response.

```
File ConCon.thy generated.
```

The result is a new Isabelle theory, which we show in Fig. 8.5.

Using the bytecode verifier `concon` annotates `Invoke` and `Throw` instructions with types narrowing their possible successors. The initial instruction of every method automatically receives an annotation restricting the types of the arguments and connecting these with the arguments lying on the stack before the call. Usually this information suffices as precondition, at least for robust methods. Finally all targets of backward jumps are annotated with $\lfloor\top\rfloor$ to signal that our VCG expects annotations there. Control constraints are essential, but usually not sufficient for verification. Other annotations, such as meaningful loop invariants, need to be added. In our example we can use `jinint` for that purpose. It runs an external interval analyser and translates its result back into annotations in our assertion language.

theory Int **imports** JBC_SafetyLogic **begin**

constdefs intervals :: pos ∼∼> expr
intervals = [
((Start,sum,7), ⌊⋀⌋ [
Rg 1 ⌊≤⌋ ⌊Intg 65535⌋,
⌊Intg 0⌋ ⌊≤⌋ Rg 2, Rg 2 ⌊≤⌋ ⌊Intg 65536⌋,
⌊Intg 0⌋ ⌊≤⌋ Rg 3, Rg 3 ⌊≤⌋ ⌊Intg 2147483647⌋])]

end

Figure 8.6: Int.thy

```
:) jinint -o Int Prog.ML
```

This results in another theory file containing interval annotations for registers.

```
File Int.thy generated.
```

As Fig. 8.6 shows, `jinint` already knows what positions our VCG expects to be annotated, such as targets of backward jumps. Instead of annotating every position, it only annotates those. For our example program we only obtain an annotation for the entry position of the loop, that is for (*Start*,*sum*,7). As Fig. 8.6 shows, this annotation limits the integers residing in registers 1, 2, and 3. Important is the upper limit for register 3. Thanks to our check against *ms* the analyser knows upper limits for both arguments of the addition. Hence, it can limit the resulting value for register 3 to 2147483647, which is $mn + ms$.

After annotations from various sources have been collected, one needs to combine these. In our example, we have to merge the control constraints from Fig. 8.5 with the intervals from Fig. 8.6. A simple way of merging two annotation maps is to conjunct them pointwise.

As Fig. 8.7 indicates, we use function *conjAn* An_1 An_2 for that purpose. After the combination Fig. 8.7 uses Isabelle's code generator to turn the final annotations into an ML format, ready for being shipped to the consumer and further processing.

8.2.3 Checking Wellformedness

Next, it is time to check the wellformedness of our program. This rules out basic errors such as type errors, missing annotations or control flow anomalies. Rather than using

theory AnProg
imports Prog ConCon Int
begin

constdefs An::pos ∼∼> expr
An = conjAn concons intervals

constdefs Π::jbc-prog
Π = (P,An)

code-module (term-of) An
file An.ML
imports VCGExecute
contains An = An

end

Figure 8.7: Packing code and annotations

our formalised wellformedness checker *wf* directly, we use a slightly modified version *wfS*. It performs exactly the same checks as *wf*, but instead of yielding *False* it gives a string with a description of the error. This helps to detect and fix wellformedness errors quickly. Note that *wfS* is also formalised in Isabelle/HOL and proven to be equivalent to *wf*.

```
:) jinwf Prog.ML An.ML
```

Our example is wellformed, hence `jinwf` accepts.

```
OK
```

8.2.4 Generating Verification Conditions

Once wellformedness has been checked, a program is ready for verification. Here our VCG comes into play. In §6.4 we have instantiated four versions of this generator, *vcg*, *vcgFrNr*, *vcgExTys* and *vcgTy*. From left to right they integrate more and more details into the formulas they produce. The first one produces the smallest conditions, but only the last has been shown to be complete. For programs with methods *vcgFrNr* is usually required. It includes the system invariant *invFrNr*, which triggers many important

simplification rules for *Catch* and *Call* expressions. Since our example does not have a functional specification to verify, *Call* and *Catch* do not matter. Hence, we can take *vcg*.

```
:) jinvcg -e "opt o vcg" Prog.ML An.ML
```

With this command we execute *vcg* on the given program and annotations, and feed the result into the optimiser *opt* from §7.3.

```
File VC.thy generated.
File Proof.thy generated.
```

The resulting verification condition is decomposed into various parts and printed into the Isabelle theory `VC.thy` shown in Fig. 8.8. Note that the original result of *vcg* has other proof obligations, one for every annotated position and the initial one. However, all these obligations are trivial and *opt* removes them. What we are left with are two conjuncts *vc-Start-sum-0* and *vc-Start-sum-7*. From the position formulas in the branch conditions the tool can automatically connect each subgoal with the method it belongs to and invent proper constant names. For example *vc-Start-main-0* stands for the conjunct of the verification condition produced for position (*Start*,*main*,*0*). Simultaneously with the condition the tool also creates `Proof.thy`, a file with an Isar proof tailored to `VC.thy`.

8.2.5 Proving the Verification Condition

The verification condition in Fig. 8.8 is structured after the control flow of the program. In a similar manner, we can structure a proof skeleton for this formula. As Fig. 8.9 shows our tool produces an Isar proof, which decomposes the entire verification condition *vc* into two subgoals, both can be handled successfully by the `clarsimp` method. For *vc-Start-sum-0* we have to verify the transitions from (*Start*,*sum*,*0*) to (*Start*,*sum*,*7*), the position with the loop invariant. To prove this goal `clarsimp` has to transform the negated $\not\geq$ relation in line 6 to the $\not\leq$ relation in line 10. The other subgoal *vc-Start-sum-7* verifies the loop body. In line 11 we first have the branch condition then the loop annotation which spans until line 13. The goal we have to show in line 25-27 is again the loop annotation, but in a transformed form. The *wpF* operator has replaced *Rg 2* with *Rg 2* $\sqcup$ *Intg 1* and *Rg 3* with *Rg 3* $\sqcup$ *Rg 2* anticipating the additions at positions (*Start*,*sum*,*19*) and (*Start*,*sum*,*15*). This explains how line 26 results from line 12 and line 27 from line 13. Line 25 contains the safety formula for the addition at position (*Start*,*sum*,*19*). Note that the original verification condition also contains such a safety formula for the addition at (*Start*,*sum*,*15*). However, this safety formula coincides with line 27, hence the optimiser has removed it. Very important

```
theory VC imports JBCSafetyLogic begin

constdefs vc-Start-sum-0 :: expr
vc-Start-sum-0 = ⌊⋀⌋ [Ty (Rg 0) (Class Start), Ty (Rg 1) Integer,
Rg 0 ⌊=⌋ (Call (St 1)), Rg 1 ⌊=⌋ (Call (St 0)), Pos (Start, sum, 0)]

⌊⇒⌋

(⌊¬⌋ (Rg 1 ⌊>⌋ ⌊Intg 65535⌋)

  ⌊⇒⌋

(Rg 1 ⌊≤⌋ ⌊Intg 65535⌋))
constdefs vc-Start-sum-7 :: expr
vc-Start-sum-7 = ⌊⋀⌋ [ Pos (Start, sum, 7), Rg 1 ⌊≤⌋ ⌊Intg 65535⌋,
⌊Intg 0⌋ ⌊≤⌋ Rg 2, Rg 2 ⌊≤⌋ ⌊Intg 65536⌋,
⌊Intg 0⌋ ⌊≤⌋ Rg 3, Rg 3 ⌊≤⌋ ⌊Intg 2147483647⌋ ]

⌊⇒⌋

(⌊¬⌋ (Rg 2 ⌊>⌋ Rg 1)

  ⌊⇒⌋

(⌊¬⌋ (Rg 3 ⌊>⌋ ⌊Intg 2147418112⌋)

    ⌊⇒⌋

(⌊⋀⌋ [(Rg 2 ⌊+⌋ ⌊Intg 1⌋) ⌊≤⌋ ⌊Intg 2147483647⌋,
⌊Intg 0⌋ ⌊≤⌋ (Rg 2 ⌊+⌋ ⌊Intg 1⌋), (Rg 2 ⌊+⌋ ⌊Intg 1⌋) ⌊≤⌋ ⌊Intg 65536⌋,
⌊Intg 0⌋ ⌊≤⌋ (Rg 3 ⌊+⌋ Rg 2), (Rg 3 ⌊+⌋ Rg 2) ⌊≤⌋ ⌊Intg 2147483647⌋])))
constdefs vc :: expr
vc = ⌊⋀⌋ [vc-Start-sum-0, vc-Start-sum-7]

lemmas vc-defs[simp] = vc-def vc-Start-sum-0-def vc-Start-sum-7-def

end
```

Figure 8.8: VC.thy

```
theory Proof
imports An Prog VC
begin

theorem vc-provable:
(P,An) ⊢ vc
proof −
have vc-holds: ∀ s. (P,An),s ⊨ vc
  proof (intro allI)
    fix s
    show (P,An),s ⊨ vc
    proof −
      have vc-Start-sum-0-holds: (P,An),s ⊨ vc-Start-sum-0
        by clarsimp

      have vc-Start-sum-7-holds: (P,An),s ⊨ vc-Start-sum-7
        by clarsimp

      from vc-Start-sum-0-holds vc-Start-sum-7-holds
      show (P,An),s ⊨ vc
        by simp
    qed
  qed
from vc-holds
show (P,An) ⊢ vc
  by (rule completeSafetyLogic)
qed

end
```

Figure 8.9: Proof.thy

are the branch conditions in lines 17 and 21. They bound the registers *Rg 2* and *Rg 3* sharply enough to bound the additions in the conclusion as requested. Again `clarsimp` can verify this formula automatically here. The reason why everything works smoothly is that we have set up the simplifier with a lot of carefully designed semantical rewrite rules for our assertion logic. These rules transform the deep embedded expressions into corresponding HOL formulas, which can then be solved by Isabelle's built in tactics and decision procedures. The arithmetic part is automatically handled by methods for Fourier Motzkin elimination [37] and Presburger arithmetics [29, 27]. In case `clarsimp` fails, the user can try to fix the proof by inserting a manual Isar script. Due to the structure and constant names of the Isar proof one knows which part of the program causes the trouble and can investigate if the failure is due to wrong annotations, a safety bug in the code or insufficiency of the proof methods.

In general the control flow graph can be quite helpful to understand and debug bytecode. For that reason we turned our control flow function *succsF* into the tool `jincfg`:

```
:) jincfg Prog.ML
```

The result is the control flow graph in a standard graph format.

```
File Prog.dot generated.
```

Using `dot` [48] one can visualise this graph. In Fig. 8.4 we show the result for our example.

8.2.6 Exporting the Proof Object

After the proof is accomplished one has various options for shipping and checking it. One method is to ship the Isar proof and let the consumer run Isabelle to check it. In this way proofs are quite small, but checking them requires heavy weight machinery. Another method is to use Isabelle's proof objects [15, 16], which are based on the Curry-Howard isomorphism. Proofs are encoded as terms whose types represent the proven theorem. Checking the proof then becomes type checking.

In the theory file shown in Fig. 8.10 we show how we export the proof of theorem *vc-provable* from Fig. 8.9 into a proof object in XML format. For the producer the proof object is the last piece of the chain. All that remains is to send the class file `Start.class`, the annotations `An.ML` and the proof object `PrfObj.xml` to the consumer.

theory PrfObj.thy **imports** Proof **begin**

```
use "ProofExporter.ML";
File.write (Path.unpack("PrfObj.xml"))
           (exportPrf (thm "vc_provable"));
```

end

Figure 8.10: Exporting the proof object

8.2.7 Checking the Proof Object

When the consumer has received the proof carrying code package, the first steps that need to be taken are equivalent to the producer. First, the consumer uses `j2jin` to obtain the Jinja representation of `Start.class`. Second, the generated ML file is checked for wellformedness together with the annotations. Third, the consumer uses the VCG to obtain the file `VC.thy`, which contains the verification condition. Fourth, the consumer runs the Isabelle proof checker to check the proof object against the freshly generated verification condition. To do this the tool `checkPrf` traverses the proof object and infers its type. This type must then be compatible with the generated verification condition. In case of a mismatch the inference is interrupted and an error message is given to the user.

```
:) checkPrf PrfObj.xml VC.thy
```

If the proof is valid, which is the case for our example, the following response is given:

```
OK
```

Now, the consumer knows that the received program is type safe, has a sound control flow and never causes an arithmetic overflow. The only parts of software it needs to trust are `j2jin`, and the proof checker `checkPrf`.

8.3 Experiments

Absence of overflows can also be tested. By placing dynamic checks in front of arithmetic operations one can modify a program such that it gives an alarm in case an overflow

N	nr. of instr.	vcg (h:m:s)	proof (h:m:s)
1	11+28	0:0:1	0:0:3
10	291	0:0:4	0:0:17
50	1411	0:0:5	0:1:19
100	2811	0:0:13	0:3:09
300	8411	0:2:23	0:12:47
500	14011	0:11:10	0:33:12

Figure 8.11: Performance: Running the VCG and constructing safety proofs.

would occur. Then one runs the programs for all inputs. Of course this is only feasible if there not many input variables involved. The example program of this chapter takes one integer, which means there are 2^{32} different inputs. A complete test run of our example (augmented with the checks) on a 3Ghz Intel Xeon with 2Gb RAM takes 1 minute and 33 seconds. With our approach, checking wellformedness, running the interval analyser and computing the verification condition takes all together 0.2 seconds. Constructing the automatic proof with Isabelle takes 3 seconds. This shows that even for small examples verification, which only abstractly executes a program, performs better than testing.

To check how our system scales we can take our example program with multiple copies of the summation method *sum1*, *sum2*, ... *sumN*. In this case the verification condition becomes bigger, because for each of the N method bodies the VCG constructs proof obligations. The formulas are the same, but our proof tactic does not exploit this. It verifies each subgoal independently and thus constructs N proofs.

In Fig. 8.11 we show the results of this experiment. It illustrates that although we just use ML code generated from an Isabelle/HOL formalisation and use a standard prover to generate proofs, our prototype scales within the expected quadratic bound (VCG).

Although the construction of the proofs is quite fast, recording and checking the proofs objects is not. For the example of this chapter this takes 10 seconds, which is still better than the full test, but a lot worse that the 3 seconds it took to construct the proof. Since theoretically proof checking is at least as efficient as proof construction, we expect much better performance by engineering the proof object module. Isabelle's proof objects tend to be very verbose.

The proof object for our simple example is already huge. Even in normalised form it has 53k nodes and requires 138kb in a zipped file. Compared to the 1.4kb of the

zipped program file with annotations, this is huge. The reason why our proofs are so big is because we use a lot of rewriting. In the current implementation of proof objects rewriting is expensive. The objects not only record the applied rule and its redex position, but also the context of the substitution. This simplifies matching in proof checking. Using oracle strings [66] to memorise the correct matchings would be much more efficient. We also observed that large parts of our proofs are due to arithmetic procedures, which cause a lot of rewriting. If we provide the arithmetic facts needed to prove the example in Fig. 8.2 as lemmas, the proof object shrinks to 7% of its original size. Engineering proof procedures towards smaller proofs [80] or using procedures tailored to the proof obligations, such as the Simplex procedure used by Necula[68], offer a lot of potential for optimisations. A rigorous approach would be to eliminate the need to record rewriting steps completely, by using reflection. In this case the proof would not record the rewriting steps, but a call to a simplification procedure that proof checker also has available. This reduces the size of proofs drastically, but comes at the price of higher complexity in the proof checker. Provided the code consumer can run Isabelle/HOL in total, one could also send the Isar proof we present in Fig. 8.9, or just the command "apply clarsimp". In general, there is a tradeoff between the size of the proof and the complexity of the system checking them.

8.4 Conclusion

The example in this section has been set up to show how Proof Carrying Code ideally works. The annotations and proofs are generated automatically from program analysers and standard proof procedures. The magic number *ms* may look a bit unnatural, but otherwise the limitations of interval analysis cannot be bridged. Since program analysis is not our main issue and more powerful analysers could be integrated just in the same manner we believe this artificial tuning does not harm our results.

In the example shown in this section, we only verify the safety policy, which is the primary concern of PCC. Our VCG and assertion logic also offer the possibility to verify functional correctness. We can annotate the `sum` method with a postcondition expressing the functionality with Gausses formula. For example, we can annotate the following formula to (*Start*,*sum*,*28*):

$(\lfloor Intg\ 2 \rfloor *\ Rg\ 3) = Rg\ 2 *\ (Rg\ 2 + \lfloor Intg\ 1 \rfloor)$
$\wedge\ Rg\ 2 = Rg\ 1 \wedge Rg\ 1 = Call\ (St\ 0)$

The proof for functional correctness is still automatic, provided we add a tactic for bounded multiplication (*mult-intervals-less* on page 142) and also use Gausses formula as loop invariant.

However, this invariant is a non-linear equation and cannot be found by the interval analyser.

Analysis techniques for polynomial relationships [65, 82] exist, but seem to be only applicable for fields rather than rings (integers). From a theoretical point of view the non-linear relationship between registers 2 and 3 could also be expressed in a linear formula. Since we have finitely many 32bit integers, we can build a formula that directly enumerates all combinations:

$(Rg\ 2 = Intg\ 0 \wedge Rg\ 3 = Intg\ 0) \vee$
$(Rg\ 2 = Intg\ 1 \wedge Rg\ 3 = Intg\ 1) \vee$
$(Rg\ 2 = Intg\ 2 \wedge Rg\ 3 = Intg\ 3) \vee \ldots$

However, the sheer size of this invariant makes this approach clearly infeasible. Testing the code on all inputs would probably be faster.

Although Isabelle produces big proofs it offers a lot of effective proof methods. In our experiments we found that `clarsimp` in combination with `arith` can prove many verification conditions automatically. Both tactics are suited perfectly for combination. The first only applies safe rules, and the latter can only be applied if it is able to finish a goal completely.
While other first order provers usually have problems with arithmetics, Isabelle/HOL offers Coopers algorithm for Presburger arithmetics [27] and other arithmetical procedures. In addition the user always has the possibility to interact, in order to prove non-linear arithmetic goals. We believe that verifying complicated programs with complex safety policy will never be fully automatable. Hence, having a system that assists a human performing that task is maybe the best we can have. For simpler properties type systems and other silent techniques should of course be preferred.

9 Conclusion

In this conclusion we shortly summarise our main achievements and experiences. We also discuss the strengths and weaknesses of our results in relation to other people's work and show ways of improvement.

9.1 Achievements

Generic Framework In the early approaches to PCC [68] (§1.3.1) the VCG was a complex, large and hardly trustable component. This thesis shows that by concentrating the essentials into a framework one can obtain a VCG that is simpler, smaller and completely verified in Isabelle/HOL.

Instantiation By instantiating our VCG to a Java-like language we demonstrate that our approach is applicable to real life languages. Although the foundations of our approach date back to Floyd [47], we had to find ways to support modern language features such as dynamic methods, objects and exception handling.

Assertion Logic The literature offers Hoare logics and assertion languages for high level languages such as Java. For the bytecode level we could not find anything like that when we started the project. Only recently bytecode logics started to emerge (see §4) and our assertion logic [97] is a contribution to that field. We have proven that first order arithmetic enriched with a few VM specific operators is enough to express annotations, weakest preconditions and safety policies for Java-like bytecode.

Advanced Safety Policies Since we use a logic instead of a specific type system our approach is very flexible in respect to the safety policies it can support. We picked arithmetic overflow, because it is practically relevant and to our knowledge not yet supported by many other tools. Interactive theorem provers such as Isabelle/HOL seem to adequate for this problem, as they provide powerful proof procedures as well as an interface to human experts. Model checkers are known to deal badly with arithmetic and testing such properties becomes quickly infeasible. Apart from certifying safety, one can also use our VCG and assertion logic to verify functional

correctness of bytecode. In contrast to Hoare Logic our verification conditions also give guarantees for non-terminating programs.

Program Analysis Many PCC systems of today are designed for a particular safety policy and embody their own program analyser or type inference algorithm. In this thesis we disconnect this problem completely from the the task of verification. We show how external analysers can be integrated and how facts coming from trusted sources can be used without having to axiomatise or re-verify them.

Prototype Using Isabelle/HOL's infrastructure, such as an ML code generator, proof objects and checkers, and proof procedures we have turned our formalisation into a runnable prototype. Apart from using this system to verify individual programs, one can see it as an example of how critical software can be developed rigorously inside a theorem prover.

9.2 Experience

In retrospect we can say that the toughest part of instantiating the framework is to get the abstract semantics right and to prove it. Also the control flow function raises difficulties in case of dynamic jumps (method invocations, exceptions).

Choosing a deep embedded assertion logic caused a lot of extra work, but our comparison [96] shows that it pays off when one is interested in smaller formulas and proofs, and wants to extract information from annotations.

Following the standard approach of stating and verifying substitution lemmas [99], worked nicely in instantiations for simpler languages [98], but in case of Jinja we gave up this idea after having almost as many different substitution lemmas as instructions.

Representing substitutions as finite maps also worked out well in simpler instantiations [96], but it was probably a mistake to keep them for Jinja. We managed to get the proofs done, but the *foldl* operator we use to construct the substitutions for the heap instructions makes the proofs for the weakest precondition operator very complicated. It would be interesting to see if a different representation of substitutions, e.g. $expr \Rightarrow expr$, simplifies these proofs.

In Fig. 9.1 we give an overview on the sizes of our Isabelle theories, they roughly coincide with the amount of time to create them. Note that this does not mention the 11kloc of

Theory, Component	Size (loc)
PCC Framework	4000
Concrete Semantics	700
Assertion Logic	5900
Control Flow	6300
Abstract Semantics	9800
Optimiser	800
System Invariants	5600
Code Lemmas	1200
Remainder	1700
total	36000

Figure 9.1: Theories and their sizes

Isabelle for the Jinja VM and its Bytecode Verifier [55]. It also does not mention the about 2kloc of Examples we have for Jinja. The previous instantiations of our framework together with their examples are also ignored [98, 96].

Our work can also be seen as a major case study for Isabelle's code generator. The entire ML code we generate from our theories is about 4kloc and turns out run stable and relatively fast. We tried to run our VCG on examples up to 10.000 bytecode instructions and up to this size it scaled well. It also turned out that HOL serves well as a modelling and programming language. One can first define functions indirectly and abstractly, prove the important properties and later on refine them to an executable style.

9.3 Discussion

Necula's pioneering PCC systems [33, 68] (see §1.3.1) are very powerful and efficient, but hardly trustworthy. One has to trust the axioms of the safety logic and the VCG, which consists of 23.000 handwritten lines of C code.

The general experience teaches that complex software and proof systems are likely to contain subtle errors. A famous example is the untyped lambda calculus, which was meant to serve as a logic, but turned out to be inconsistent. Hence, it is not surprising that the *Special J* system [33], which is closely related to Touchstone (see §1.3.1), has been reported to have holes in its safety logic. These can be exploited to smuggle unsafe code through the checks [57].

Since also a bug in the implementation of a VCG can lead to a proof of the wrong formula a new approach to PCC came up.

Appel claims, "Constructing a mechanically-checkable correctness proof of a VCGen would be a daunting task" ([6] p.2), and proposes foundational PCC (FPCC) (see §1.3.2). In FPCC the VCG and other untrusted components, such as a safety logic based on obscure axioms, are removed and just the proof checker and the formalised machine semantics remain.

The idea is to formulate a safety theorem directly on the semantics, e.g. *isSafe* Π, and submit a proof for it. For constructing the proof all means (type system, VCG, ...) are permitted, but these must be part of the proof. This makes FPCC very flexible, but as we point out in §1.3.2 it is hard to construct foundational proofs automatically and in compact form.

New publications on FPCC deviate from the original intention and use type systems with verified rules [31]. Type rules are closely related with an abstract semantics and many type systems [36, 35] are so expressive that they are very similar to an assertion logic. In that respect FPCC has moved towards the conventional PCC setup.

On the other hand Necula's group has moved towards FPCC with the Open Verifier approach (see §1.3.6). There the VCG's trustworthiness is increased by reducing it to a small core that cooperates with extensions that are obliged to justify their work with proofs.

Our approach has many similarities with the Open Verifier. The generic VCG can be seen as a core and the parameters together with the program analysers as extensions. However, there is a fundamental difference. Our VCG and the parameters come with machine-checkable proofs. This means, we can trust the work of our VCG and its parameters. Hence, we do not have to include additional certificates in our safety proofs.

One can see our approach as a union of conventional and foundational PCC. We use a VCG, a logic and an abstract semantics just as conventional PCC does, but we have verified these and thus have reduced the trusted code base to the checker and machine model, just as FPCC does. Our generated prototype does not offer the efficiency of systems like Touchstone, but one has to take into account that it can verify non-standard safety policies and even functional correctness.

9.4 Further Work

The PCC system we presented in this thesis can be improved in a number of ways.

More Bytecode The Jinja language already contains a representative subset of real Java bytecode, but some practically important concepts are missing. For example we ignored non-default constructors, arrays, and threads and all the remaining

primitive datatypes, e.g. `float`, `byte`, `string`. In the conclusion of §3 we pointed out that apart from threads integrating these features is mainly an engineering task.

More Assertions Apart from the bytecode, also our assertion logic has shortcomings. The fact that it is first order clearly limits its expressibility when it comes to pointer structures. Transitive closures and reachability cannot be expressed in the current version. Here, we plan to integrate primitive recursion. This also allows to integrate user defined functions into annotations without risking consistency. Since Isabelle/HOL also supports primitive recursion, we should be able to support it with our approach to verify formulas semantically by translating them to HOL.

More Analysers With an extended assertion logic we plan to integrate other analysers for Java bytecode. In particular, the object oriented aspect is only marginally supported with our existing analysers. The bytecode verifier only infers the types of references leading out of the current frame. Identifying shapes in the reference structure is clearly beyond its capability. We believe, that with the integration of pointer analysis tools, we can support other safety policies such as excluding null pointer dereferences.

Less Proof Objects Another desirable goal are smaller proof objects. Since rewriting really boosts them up, reflection should be integrated into Isabelle/HOL. This would immediately eliminate the need to justify rewriting, but comes with the price of more complexity in the proof checker.

A Appendix

A.1 Isabelle/HOL

This chapter gives a short overview on the notation of Isabelle/HOL, which we use as formal meta-language throughout this thesis.

A.1.1 Types

The basic types of truth values, natural numbers and integers are called *bool*, *nat*, and *int*. The space of total functions is denoted by $\Rightarrow$. Type variables are written $'a$, $'b$, etc. The notation $t::\tau$ means that HOL term t has HOL type τ. Every type in Isabelle/HOL has at least one element. The element *arbitrary* is known to exist in any type, but comparisons with "ordinary" elements of each type cannot be decided within *Isabelle/HOL*. Hence, *arbitrary* behaves like a free variable.

A.1.2 Pairs

The type $'a \times 'b$ denotes all pairs of elements from $'a$ and $'b$. Pairs come with the two projection functions $fst :: 'a \times 'b \Rightarrow 'a$ and $snd :: 'a \times 'b \Rightarrow 'b$. We identify tuples with pairs nested to the right: (a, b, c) is identical to $(a, (b, c))$ and $'a \times 'b \times 'c$ is identical to $'a \times ('b \times 'c)$.

A.1.3 Sets

Sets (type $'a\ set$) follow the usual mathematical convention. For example $(nat \times nat)\ set$ is the set of all pairs of natural numbers, i.e. $\{(0,0),(0,1),(1,0),(0,2),\ldots\}$. With $\{x.\ P\ x\}$ we denote the set of all x satisfying P.

A.1.4 Lists

Lists (type *'a list*) come with the empty list [], the infix constructor $\cdot$, the infix @ that appends two lists, and the conversion function *set* from lists to sets. For example, we have $1 \cdot [2,3] = [1,2,3] = [1,2]@[3]$ and *set* $[1,2,3] = \{1,2,3\}$. The destructors *hd* and *tl* yield the first element of a list and its tail, e.g. *hd* $[1,2,3] = 1$ and *tl* $[1,2,3] = [2,3]$. Likewise, the destructors *last* and *butlast* , yield the last element and the list before it, e.g. *last* $[1,2,3] = 3$ and *butlast* $[1,2,3] = [1,2]$. To split lists we have *take n xs* and *drop n xs* . The first yields the list up to the nth element and the latter the list after it, e.g. *take 1* $[1,2,3] = [1]$ and *drop 1* $[1,2,3] = [2,3]$. Variable names ending in "s" usually stand for lists, $|xs|$ is the length of *xs*, and $xs_{[n]}$, where n::*nat*, is the nth element of *xs* (starting with 0), e.g. $|[1,\ 2,\ 3]| = 3$ and $[1,2,3]_{[1]} = 2$. The notation $[i..<j]$ with i::*nat* and j::*nat* stands for the list $[i, \ldots, j-1]$. The predicate *distinct xs* means that the elements of *xs* are all distinct. Finally, we have the standard functions *map f xs* and *filter P xs* , also writen as $[x{\in}xs.\ P\ x]$. For example, take *map fst* $[(1,2),(4,3)] = [1,4]$ and $[(x,y) \in [(1,2),(4,3)].\ x > y] = [(4,3)]$.

A.1.5 Option

The type *'a option*

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

adjoins a new element *None* to a type *'a*. For succinctness we write $\lfloor a \rfloor$ instead of *Some a*. The under-specified inverse *the* of *Some* satisfies *the* $\lfloor x \rfloor = x$.

A.1.6 Functions

Isabelle/HOL is a logic of total functions only. Functions are typically declared by with a name and a type. For example $f :: nat \Rightarrow nat$ declares a total function over the natural numbers. Functions are usually defined via equations, where the left hand side contains the function symbol and a list of parameters and the right hand side a term with already defined functions. For example

$f\ x = x{*}x$

defines f to be the square function. Another possibility is to define functions with the λ operator:

$f = \lambda\ x.\ x{*}x$

Finally, there is the possibility to define functions via recursive equations. Note that

Isabelle/HOL provides different methods to define recursive functions and some require to prove termination explicitly. However, for the sake of simplicity, we do not distinguish these variants and just write the equations. For example the following equations define the square function recursively:

$f\ 0 = 0$
$f\ (Suc\ n) = f\ n + n + n - 1$

Note that in case one does not list equations matching all input combinations, Isabelle/HOL automatically adds equations yielding *arbitrary* in order to obtain a total definition. The *the* operator above is an example for that. Isabelle/HOL internally generates *the None = arbitrary*, where the type of *arbitrary* depends on the context *the* is used.

Functions can be composed via the o operator. For example $(f\ o\ g)\ x$ is the same as $f\ (g\ x)$.

Function update is written $f(x := y)$ where $f :: 'a \Rightarrow 'b$, $x :: 'a$ and $y :: 'b$.

Partial functions can be modelled as functions of type $'a \Rightarrow 'b\ option$, where *None* represents undefinedness and $f\ x = \lfloor y \rfloor$ means x is mapped to y. For updating partial functions we sometimes write $f(x \mapsto y)$ as a shorthand for $f(x{:=}\lfloor y \rfloor)$

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

map-of $[] = empty$
map-of $(p \cdot ps) = map\text{-}of\ ps(fst\ p \mapsto snd\ p)$

A.1.7 Finite Maps

In addition to functional maps ($'a \Rightarrow 'b\ option$), we also use finite maps . They have type $'a \sim\!\!> 'b$, which is just an abbreviation for association lists $('a \times 'b)\ list$. Finite maps are lists of pairs, e.g. $fm = [(1,1),(3,5),(3,6)]$, and have operations for lookup, e.g. $fm\ ?\ 0 = None$ or $fm\ ?\ 3 = \lfloor 5 \rfloor$, domain, e.g. $dom\ fm = [1,3,3]$ and range, e.g. $ran\ fm = [1,5,5]$. Note that a pair (x,y) is overwritten by a pair (x,y') to the left of it. To combine finite maps one can use ordinary list concatenation, e.g. $(fm@[4,6])\ ?\ 4 = \lfloor 6 \rfloor$. Note that the lookup operator $?$ is just an infix notation for the *map-of* operator above, i.e. $fm\ ?\ x = map\text{-}of\ fm\ x$. The advantage finite maps have over functional ones is that we can induct on their representation (lists) and that *dom* and *ran* are computable.

A.1.8 Locales

Locales [13] are detached proof contexts. They support modular reasoning inside Isabelle theories. In this thesis we use locales to define an abstract framework for our VCG. Locales declare parameters and requirements on these. Inside a locale these parameters can be used to define further functions or to state theorems, whose proofs can assume all the requirements. To make these theorems available for the outside world, one has to instantiate a locale. This is done by providing defined functions for the parameters and by proving that these satisfy the requirements. In contrast to theories all type variables used inside locales must be instantiated with the same type. For example in locale *Semantics* on page 24 both parameters *initS* and *effS* must be instantiated to a common type for $'pos \times 'mem$.

A.2 Additional Definitions

A.2.1 External VCG

In this section we define *isafe* and *vcG*, which are used in the locale *VCG* to construct inductive safety formulas and generic verification conditions. Since locales do not allow recursive definitions, we have to define these operators outside, with the drawback that all parameters declared inside the locale *VCG* must be given as additional arguments. Since type variables can be renamed as one likes we use $'p$, $'P$ and $'f$ as shorthands for $'pos$, $'prog$ and $'form$.

$isafe::('p\ list \times 'P \times ('P \Rightarrow 'p \Rightarrow 'f\ option) \times 'p \times 'f \times ('f\ list \Rightarrow 'f) \times ('f \Rightarrow 'f \Rightarrow 'f) \times ('P \Rightarrow 'p \Rightarrow 'f) \times (\ 'P\ \Rightarrow 'p \Rightarrow ('p \times 'f)\ list) \times (\ 'P\ \Rightarrow 'p \Rightarrow 'p \Rightarrow 'f \Rightarrow 'f)) \Rightarrow 'f$

$isafe\ (S,\ \Pi,\ anF,\ p,\ \lfloor F \rfloor,\ \lfloor\bigwedge\rfloor,\ \lfloor\Rightarrow\rfloor,\ safeF,\ succsF,\ wpF) =$
if $p \notin set\ S$ *then* $\lfloor F \rfloor$
else $\lfloor\bigwedge\rfloor\ [safeF\ \Pi\ p]\ @\ (case\ anF\ \Pi\ p$
of None $\Rightarrow map\ (\lambda\ (p',B).\ B\ \lfloor\Rightarrow\rfloor$
$wpF\ \Pi\ p\ p'\ (isafe\ ([q{\in}S.\ q \neq p],\Pi,anF,\ p,\ \lfloor F \rfloor,\ \lfloor\bigwedge\rfloor,\ \lfloor\Rightarrow\rfloor,\ safeF,\ succsF,\ wpF)))$
$(succsF\ \Pi\ p)$
$\mid \lfloor A \rfloor \Rightarrow [A])$

By setting S to the full code domain $domC\ \Pi$ and passing the corresponding parameters locale *VCG* defines *isafeF* as follows:

$isafeF = (\lambda\ \Pi\ p.\ isafe\ (domC\ \Pi,\ \Pi,\ anF,\ p,\ \lfloor F \rfloor,\ \lfloor\bigwedge\rfloor,\ \lfloor\Rightarrow\rfloor,\ safeF,\ succsF,\ wpF))$

Analogously we define *vcG* for the generic verification condition generator.

$$vcG :: ('f\ list \Rightarrow 'f) \Rightarrow ('f \Rightarrow 'f \Rightarrow 'f) \Rightarrow 'f \Rightarrow ('P \Rightarrow 'p) \Rightarrow ('P \Rightarrow 'f) \Rightarrow ('P \Rightarrow 'p \Rightarrow 'f) \Rightarrow ('P \Rightarrow 'p \Rightarrow ('p \times 'f)\ list) \Rightarrow ('P \Rightarrow 'p \Rightarrow 'p \Rightarrow 'f \Rightarrow 'f) \Rightarrow ('P \Rightarrow 'p\ list) \Rightarrow ('P \Rightarrow 'p\ list) \Rightarrow ('P \Rightarrow ('p \Rightarrow 'f\ option)) \Rightarrow 'P \Rightarrow 'f$$

$$\begin{array}{l}
vcG\ \llcorner\bigwedge\lrcorner\ \llcorner\Rightarrow\lrcorner\ \llcorner F\lrcorner\ ipc\ initF\ safeF\ succsF\ wpF\ domC\ domA\ anF\ \Pi = \\
(\textsf{let}\ isafeF = (\lambda\ \Pi\ p.\ isafe(domC\ \Pi,\Pi,anF,p,\llcorner F\lrcorner,\llcorner\bigwedge\lrcorner,\llcorner\Rightarrow\lrcorner,safeF,succsF,wpF)) \\
\textsf{in}\ (\llcorner\bigwedge\lrcorner\ [initF\ \Pi\ \llcorner\Rightarrow\lrcorner\ (isafeF\ \Pi\ (ipc\ p))]\ @ \\
\qquad (map(\lambda\ p_a.\ \llcorner\bigwedge\lrcorner\ (map(\lambda\ (p',B).\ (\llcorner\bigwedge\lrcorner\ [isafeF\ \Pi\ p,\ B]\ \llcorner\Rightarrow\lrcorner \\
\qquad\qquad\qquad\qquad (wpF\ \Pi\ p\ p'\ (isafeF\ \Pi\ p')))) \\
\qquad\qquad\qquad\qquad (succsF\ \Pi\ p))) \\
\qquad\qquad (domA\ \Pi))))
\end{array}$$

By passing *vcG* all parameters the locale VCG defines the function *vcg*.

$$vcg = (\lambda\ \Pi.\ vcG\ \llcorner\bigwedge\lrcorner\ \llcorner\Rightarrow\lrcorner\ \llcorner F\lrcorner\ ipc\ initF\ safeF\ succsF\ wpF\ domC\ domA\ anF\ \Pi)$$

Note that under the assumption that Π is wellformed, we can derive the simpler definitions for *isafeF* and *vcg* shown in Fig. 2.5 and Fig. 2.6.

A.2.2 Wellformedness

This section defines the function *checkPos*, which performs many of the wellformedness checks discussed in §6.2.

$$\begin{array}{l}
checkPos :: jbc\text{-}prog \Rightarrow (pos\ list) \Rightarrow bool \\
checkPos\ \Pi\ [] = True \\
checkPos\ \Pi\ (p \cdot ps) = \\
(\textsf{if}\ (\textsf{let}\ scsn = map\ fst\ (succsNormal\ \Pi\ p); \\
\qquad scse = map\ fst\ (succsExcept\ \Pi\ p) \\
\quad \textsf{in}\ list\text{-}all \\
\qquad (\lambda p'.\ (idx\ (domC\ \Pi)\ p' \leq idx\ (domC\ \Pi)\ p \longrightarrow anF\ \Pi\ p' \neq None) \wedge \\
\qquad\quad p'\ mem\ domC\ \Pi \wedge \\
\qquad\quad (p'\ mem\ scsn \longrightarrow handlesEx\ (fst\ \Pi)\ p' = None) \wedge \\
\qquad\quad p' \neq ipc\ \Pi) \\
\qquad (scsn\ @\ scse) \wedge \\
\qquad set\ scse \subset set\ (domC\ \Pi)) \wedge \\
\quad throwChk\ (\Pi,\ cmd\ \Pi\ p,\ anF\ \Pi\ p,\ p) \wedge \\
\quad invokeChk\ (\Pi,\ cmd\ \Pi\ p,\ anF\ \Pi\ p,\ p) \\
\textsf{then}\ checkPos\ \Pi\ ps\ \textsf{else}\ False)
\end{array}$$

The definition above uses two auxiliary functions to check `Invoke` and *Throw* instructions

```
invokeChk :: jbc-prog × (instr option) × (expr option) × pos ⇒ bool
invokeChk =
λ(Π, ins, an, p).
  case ins of None ⇒ True
  | ⌊c⌋ ⇒
    case c of
    Invoke M n ⇒
      case an of None ⇒ False
      | ⌊A⌋ ⇒
        if p = ipc Π then False
        else case extractTy (A, St n) of [] ⇒ False
             | ty · tys ⇒
               list-all
                (ty-case False False False True
                  (λX. JBC-Semantics.has-method (fst Π) X M))
                (ty · tys) ∧
               M ≠ fst (snd (ipc Π))
    | - ⇒ True throwChk :: jbc-prog × (instr option) × (expr option) × pos ⇒ bool
throwChk =
λ(Π, ins, an, p).
  case ins of None ⇒ True
  | ⌊c⌋ ⇒
    case c of
    Throw ⇒ case an of None ⇒ False
            | ⌊A⌋ ⇒
              if p = ipc Π then False
              else case extractTy (A, St 0) of [] ⇒ False
                   | ty · tys ⇒
                     list-all
                      (ty-case False False False False (λX. True))
                      (ty · tys)
    | - ⇒ True
```

A.3 Additional Proofs

A.3.1 Instantiating the Abstract Semantics

One of the more complicated cases is the normal behaviour of `Putfield` F C. We have a closer look on this case now, as it reveals a lot about what needs to be proven in all the other cases as well.

assumes
cmd-p: $cmd\ \Pi\ p = \lfloor i \rfloor$
i-def: $i = Putfield\ F\ Cl$
handlesEx: $handlesEx\ (fst\ \Pi)\ p' = None$
p-def: $p = (C,M,pc)$
sigma-def: $\sigma = (None,h,(stk,loc,p) \cdot frs)$
sigma'-def: $\sigma' = (None,h',fr' \cdot frs')$
e'-def: $e' = e(\!|cs := \mathsf{if}\ \exists M\ n.\ i = Invoke\ M\ n\ \mathsf{then}\ h\ \cdot\ cs\ e\ \mathsf{else\ if}\ i = Return\ \mathsf{then}\ tl\ (cs\ e)\ \mathsf{else}\ cs\ e|\!)$
p'-def: $p' = snd\ (snd\ fr')$
check-i: *check-instr'* $i\ P\ h\ stk\ loc\ C\ M\ pc\ frs$
exec-i : *exec-instr* $i\ P\ h\ stk\ loc\ C\ M\ pc\ frs = \sigma'$

shows
G: $\forall\ I.\ evalE\ \Pi\ (p,\sigma,e(\!|lv:=I|\!))\ (wpF\ \Pi\ p\ p'\ Q) = evalE\ \Pi\ (p',\sigma',e'(\!|lv:=I|\!))\ Q$
proof (*cases* $hd\ (tl\ stk) = Null$)

First, of all we check whether there is a *NullPointer* on top of the stack. If so, *effS* raises an exception and sets the exception flag. Since σ' has an unset exception flag we have a contradiction and are done with proving this case.

case *True*: $hd\ (tl\ stk) = Null$

from *True exec-i i-def sigma'-def*

show G

Next comes the case with no *Null* reference.

case *False*: $hd\ (tl\ stk) \neq Null$

Here *effS* executes normally and we can derive more details about the states. The successor frame has a shorter operand stack.

from *False i-def exec-i sigma-def sigma'-def*
have *Afr'*: $fr' = (tl\ (tl\ stk),\ loc,\ C,\ M,\ Suc\ pc)$

Only the topmost frame changes.

from *False i-def exec-i sigma-def sigma'-def*
have *Afrs'*: $frs' = frs$

From *check-i* we get that there are at least two elements *st* and *st'* on top of the stack.

from *False i-def exec-i sigma-def check-i*
obtain *st st' stk'* **where** *Astk*: $stk = st \cdot (st' \cdot stk')$

We also get that st' is a reference r to some object ob.

from *False Astk i-def exec-i sigma-def sigma'-def check-i*
obtain *r ob* **where** *Ar-ob*: $st' = Addr\ r \wedge h\ r = \lfloor ob \rfloor$

The environment does not change at all.

from *False i-def exec-i sigma-def sigma'-def Astk e'-def*
have *Ae'*: $e' = e$

In successor heap h' field F of object ob has been updated with st.

from *False i-def exec-i sigma-def sigma'-def Astk Ar-ob*
have *Ah'*: $h' = h(r \mapsto (cname\text{-}of\ h\ r, (snd\ ob)((F, Cl) \mapsto st)))$

The program counter is incremented by one.

from *Afr' p'-def*
have *p'-def* : $p' = (C,M,Suc\ pc)$

Now, we start looking the symbolic manipulations. For `Putfield` $F\ Cl$ we get the following substitution map mp. We decompose it to a basic map em, which substitutes Pos and St expressions, and a map for heap expressions obtained by folding f on the list of address subexpressions $remdup\ (getGfEx\ F\ Cl\ Q)$ in Q.

obtain *em* **where** *Aem*:
$em = map\ (\lambda q.\ (Pos\ q,\ \mathsf{if}\ q = p'\ \mathsf{then}\ Pos\ p\ \mathsf{else}\ \lfloor F \rfloor))\ (getPosEx\ Q)\ @$
$\quad (map\ (\lambda k.\ (St\ k,\ St\ (Suc\ (Suc\ k))))\ (stkIds\ Q))$

obtain *f* **where** *Af*:
$f = (\lambda mp\ ex.\ let\ ex' = substE\ mp\ ex$
$\quad in\ (Gf\ F\ Cl\ ex,$
$\quad\ \lfloor if \rfloor\ ex' \sqsubseteq (St\ (Suc\ 0))\ \lfloor then \rfloor\ St\ 0$
$\quad\ \lfloor else \rfloor\ Gf\ F\ Cl\ ex')\ \cdot\ mp)$

obtain *mp* **where** *Amp*:
$mp = foldl\ f\ em\ (remdup\ (getGfEx\ F\ Cl\ Q))$

To handle lookups in mp, we have to get through the *foldl* operator. For this reason, we frequently use the lemma *foldl-map-lookup*, which moves the lookup operator behind the *foldl* whenever the expression we lookup cannot match a maplet f constructs.

have *foldl-map-lookup*:

$\bigwedge es\ x.\ (\forall ex \in set\ es.\ x \neq Gf\ F\ Cl\ ex) \Longrightarrow \forall mp'.\ foldl\ f\ mp'\ es\ ?\ x = mp'\ ?\ x$

Before we start an induction on Q, we introduce an alias Q' for it. With that we distinguish the expression being substituted (Q' from now on) from the one used to extract patterns for the map *mp*.

obtain Q' **where** AQ': $Q = Q'$

From that we can easily derive that Q' is also a subexpression of Q.

from AQ'
have Q'-*subExpr-Q*: $Q' \in set\ (subExpr\ Q)$

This relationship is weaker, but more stable for induction on Q', which follows now. We perform this induction on goal *G2*, which uses $Q' \in set\ (subExpr\ Q)$ as premise. This is important as the substitution map *mp* only has entries for subexpressions of Q. We end up with 19 different cases for Q'. For brevity's sake, we omit the proofs for most cases below. Instead, we illustrate the proof on a few cases only, including the complicated one where $Q' = Gf\ F\ Cl\ ex$.

have *G2*:$\bigwedge I.\ Q' \in set\ (subExpr\ Q) \Longrightarrow$
$(evalE\ \Pi\ (p,\ \sigma,\ e(\!|lv := I|\!))\ (substE\ mp\ Q') = evalE\ \Pi\ (p',\ \sigma',\ e'(\!|lv := I|\!))\ Q')$
proof (*induct* Q' *rule*: *expr-induct*)

Let us start with an easy case, $Q' = Rg\ k$. From the premise we obtain *subEx*, which says that $Rg\ k$ occurs in Q.

assume *subEx*: $Rg\ k \in set\ (subExpr\ Q)$

Obviously $Rg\ k$ is not an heap expression.

have *neqGf*: $\forall\, ex \in set\ (remdup\ (getGfEx\ F\ Cl\ Q)).\ Rg\ k \neq Gf\ F\ Cl\ ex$

Since *em* does not substitute $Rg\ k$ expressions, we can use the *foldl-map-lookup* on the fact above and obtain that $Rg\ k$ is not modified by the substitution.

from *Amp Af Aem neqGf foldl-map-lookup*
have *AsubstE-mp*: $substE\ mp\ (Rg\ k) = Rg\ k$

Since `Putfield` $F\ Cl$ does not touch registers either, we can finish the case by evaluating both expressions.

from *AsubstE-mp Afr' sigma-def sigma'-def foldl-map-lookup*
show $evalE\ \Pi\ (p,\ \sigma,\ e(\!|lv := I|\!))\ (substE\ mp\ (Rg\ k)) =$
$evalE\ \Pi\ (p',\ \sigma',\ e'(\!|lv := I|\!))\ (Rg\ k)$

Next, we have $Q' = St\ k$. Since $St\ k$ is not an heap expression, we can use *foldl-map-lookup* to turn $substE\ mp\ (St\ k)$ into $substE\ em\ (St\ k)$.

assume *subEx*: $St\ k \in set\ (subExpr\ Q)$

have *neqGf*: $\forall\, ex \in set\ (remdup\ (getGfEx\ F\ Cl\ Q)).\ St\ k \neq Gf\ F\ Cl\ ex$
from *neqGf subEx Amp Af foldl-map-lookup*
have *AsubstE-mp*: $substE\ mp\ (St\ k) = substE\ em\ (St\ k)$

To find the maplet *em* provides for $St\ k$, we only have to show that k is among the extracted stack indices. This follows from *subEx*.

from *subEx*
obtain *sid sid'* **where** *AIds*: $stkIds\ Q = sid\ @\ (k \cdot sid') \wedge k \notin set\ sid$

From this and the facts we derived about the effects `Putfield` $F\ Cl$ has on the states, we can finish this case.

from *AsubstE-mp Aem Afr' sigma-def sigma'-def Astk AIds*
show $evalE\ \Pi\ (p,\ \sigma,\ e(\!|lv := I|\!))\ (substE\ mp\ (St\ k)) =$
$evalE\ \Pi\ (p',\ \sigma',\ e'(\!|lv := I|\!))\ (St\ k)$

With $NewA\ n$ we have a subexpression of Q that accesses the heap.

assume *subEx*: $NewA\ n \in set\ (subExpr\ Q)$

Since, $NewA\ n$ does not match any Gf expression, we can derive *neqGf* and use *foldl-map-lookup* to obtain *AsubstE-mp*.

have *neqGf*: $\forall\, ex \in set\ (remdup\ (getGfEx\ F\ Cl\ Q)).\ NewA\ n \neq Gf\ F\ Cl\ ex$

from *Amp neqGf Aem Af foldl-map-lookup*
have *AsubstE-mp*: $substE\ mp\ (NewA\ n) = NewA\ n$

As *AsubstE-mp* shows the substitution does not modify $NewA\ n$. This means that `Putfield` $F\ Cl$ does not affect it either. This is the case as the reference $NewA\ n$ yields depends only on the references currently allocated with objects, but not on their field values.

from *subEx AsubstE-mp Afr' sigma-def sigma'-def Ah' Astk Ar-ob*
show $evalE\ \Pi\ (p,\ \sigma,\ e(\!|lv := I|\!))\ (substE\ mp\ (NewA\ n)) =$
$evalE\ \Pi\ (p',\ \sigma',\ e'(\!|lv := I|\!))\ (NewA\ n)$

With an $Gf\ F'\ Cl'\ ex$ expression things become complicated. Here we have to deal with the *foldl* operator in *mp*.

assume *subEx*: $Gf\ F'\ Cl'\ ex \in set\ (subExpr\ Q)$

Since $Gf\ F'\ Cl'\ ex$ is a composed expression, we get an induction hypothesis for the smaller expression *ex*.

assume *hyp*: $\bigwedge I.\ ex \in set\ (subExpr\ Q) \Longrightarrow$
$evalE\ \Pi\ (p,\ \sigma,\ e(\!|lv := I|\!))\ (substE\ mp\ ex) =$

$evalE\ \Pi\ (p', \sigma', e'(\!|lv := I|\!))\ ex$

The premise for *hyp* follows easily from *subEx* and the fact that *subExpr* is transitive.

from *subEx*
have *ex-subEx*: $ex \in set\ (subExpr\ Q)$

This enables us to split the list of address expressions used at the very bottom of *mp*, into the sublists *as*, [*ex*] and *bs*.

from *subEx*
have *ex-getGfEx*: $ex \in set\ (getGfEx\ F'\ Cl'\ Q)$

from *ex-getGfEx* **obtain** *as bs*
  where *Aasbs*: $remdup\ (getGfEx\ F'\ Cl'\ Q) = as\ @\ (ex\ \#\ bs)$
    $\wedge\ ex \notin set\ as \wedge\ ex \notin set\ bs$

We start proving the goal by a case distinction on $F = F'$ and $C = Cl$.

show *GGf*: $evalE\ \Pi\ (p, \sigma, e(\!|lv := I|\!))\ (substE\ mp\ (Gf\ F'\ Cl'\ ex)) =$
    $evalE\ \Pi\ (p', \sigma', e'(\!|lv := I|\!))\ (Gf\ F'\ Cl'\ ex)$
  proof $(cases\ F = F' \wedge Cl = Cl')$

case *False*: $F \neq F' \vee Cl \neq Cl'$

In this case the `Putfield` $F\ C$ instruction modifies a different field than the one accessed by $Gf\ F'\ Cl'\ ex$. With *neqGf* and *em-lup* we prepare ourselves to use *fold-map-lookup* in order to show in *AsubstE-mp* that substitution simply moves into $Gf\ F'\ Cl'\ ex$.

from *False*
have *neqGf*:
$\forall\, ex' \in set\ (remdup\ (getGfEx\ F\ Cl\ Q)).\ Gf\ F'\ Cl'\ ex \neq Gf\ F\ Cl\ ex'$

from *Aem*
have *em-lup*:
  $em\ ?\ Gf\ F'\ Cl'\ ex = None$

from *False Af Amp Aem em-lup neqGf foldl-map-lookup*
have *AsubstE-mp*: $substE\ mp\ (Gf\ F'\ Cl'\ ex) = (Gf\ F'\ Cl'\ (substE\ mp\ ex))$

With *AsubstE-mp* we can finish the goal by evaluating both sides using the induction hypotheses, to obtain $evalE\ \ldots\ (substE\ mp\ ex) = evalE\ \ldots\ ex$.

from *False AsubstE-mp Afr' sigma-def sigma'-def Ah'*
  Astk Ar-ob p'-def Aob hyp[*of I*] *ex-subEx*

show *evalE* Π (*p*, σ, *e*(|*lv* := *I*|)) (*substE mp* (*Gf F′ Cl′ ex*)) =
evalE Π (*p′*, σ′, *e′*(|*lv* := *I*|)) (*Gf F′ Cl′ ex*)

case *True*: $F = F' \wedge Cl = Cl'$

Now comes the complicated case, where the expression *Gf F′ Cl′ ex* may or may not access the modified field.

We start with deriving *neqGf-as* and *neqGf-bs*, which we need for *foldl-map-lookup* later on.

from *Aasbs*
have *neqGf-as*: $\forall\, exp \in set\ as.\ Gf\ F\ Cl\ ex \neq Gf\ F\ Cl\ exp$

from *Aasbs*
have *neqGf-bs*: $\forall\, exp \in set\ bs.\ Gf\ F\ Cl\ ex \neq Gf\ F\ Cl\ exp$

We can alter the substitution map *mp* to the following pattern with nested *foldl*.

from *True Amp Aasbs*
have *Amp*: *mp* = *foldl f* (*f* (*foldl f em as*) *ex*) *bs*

Inside this pattern we have *mp1*, a map that only substitutes expressions *getGfEx* extracts before it detects *ex*. Note that this already contains all subexpressions of *ex* that also occur in a *Gf F′ Cl′* context.

obtain *mp1* **where** *Amp1*:
mp1 = (*foldl f em as*)

This is why we can use *mp1* to modify the address expression *ex*. In *mp2* we apply the *f* in *mp* onto *ex*.

obtain *mp2* **where** *Amp2*:
mp2 = (*foldl f* ((*Gf F Cl ex*,*let ex′* = *substE mp1 ex*
in (*if ex′* = *St* (*Suc 0*) *then St 0*
else Gf F Cl ex′)) · *mp1*) *bs*)

We can show that *mp2* is just an unfolded version of *mp*.

from *Amp Amp1 Amp2*
have *Amp*: *mp* = *mp2*

Since *mp1* already contains the maplets for all subexpressions of *ex*, the substitution maps *mp2* and *mp1* amount to the same when we apply them to *ex*

have *substE-mp2-mp1*: *substE mp2 ex* = *substE mp1 ex*
proof −

However, showing this relationship is tricky and requires another induction. Before we come to that, we introduce the fact *eqExMps-mp2-mp1*, which claims that the expressions maps *mp2* and *mp1* are equally suited for substitution on *ex*.

eqExMps :: (*expr* ∼∼> *expr*) ⇒ (*expr* ∼∼> *expr*) ⇒ *expr* ⇒ *bool*

eqExMps em em′ ex = *foldE* (λ(*ex*, *a*). *em* ? *ex* = *em′* ? *ex* ∧ *list-all* (λ*x*. *x*) (*noCC* (*ex*, [*a*]))) *op* ∧ *True ex*

have *eqExMps-mp2-mp1*: *eqExMps mp2 mp1 ex*
proof −

Before we induct, we introduce an alias for *ex* just as we did for *Q*.

obtain *ex′* **where** *Aex′*: *ex* = *ex′*
  from *Aex′*
have *Aex′-subEx*: *ex′* ∈ *set* (*subExpr ex*)

Then we prove *eqExMps-ind* by induction on *ex′*. This gives another 19 cases, which we omit here for the sake of brevity.

have *eqExMps-ind*:
  ex′ ∈ *set* (*subExpr ex*) ⟹ *eqExMps mp2 mp1 ex′*

from *Aex′ Aex′-subEx eqExMps-ind*
show *eqExMps mp2 mp1 ex*
  by *simp*
qed*(eqExMps-mp2-mp1)*

Note that *eqExMps* is designed to guarantee equivalence on substituion. Hence, we can conclude our initial goal.

from *eqExMps-mp2-mp1*
  show *substE mp2 ex* = *substE mp1 ex*
 qed*(substE-mp2-mp1)*

Now, we have everything to instantiate our hypotheses *hyp* with *mp1* instead of *mp*.

from *hyp ex-subEx Amp substE-mp2-mp1*
have *evalE-ex*: *evalE* Π (*p*, *σ*, *e*⦇*lv* := *I*⦈) (*substE mp1 ex*) = *evalE* Π (*p′*, *σ′*, *e′*⦇*lv* := *I*⦈) *ex*

With that modified hypothesis, we can now finish the case by evaluating both sides.

from *True Amp substE-mp2-mp1 Amp2 neqGf-bs Af evalE-ex*
    sigma′-def sigma-def Astk Ah′ Ar-ob Aob Afr′
show *evalE* Π (*p*, *σ*, *e*⦇*lv* := *I*⦈) (*substE mp* (*Gf F′ Cl′ ex*))

$= evalE\ \Pi\ (p',\ \sigma',\ e'(\!|lv := I|\!))\ (Gf\ F'\ Cl'\ ex)$
qed(GGf)

The remaining cases can be handled analogously to the previous ones. We do not have any further expressions accessing the heap. Hence, all these cases are as simple as the ones for *Rg k*.

qed*(G2)*

Now, we can use our proven goal *G2* to obtain our initial one, *G*. Since $Q = Q'$ this adaption is trivial.

from *G2 Q'-subExpr-Q AQ' Amp Aem Af False handlesEx i-def cmd-p*
show
$\forall I.\ evalE\ \Pi\ (p,\ \sigma,\ e(\!|lv := I|\!))\ (wpF\ \Pi\ p\ p'\ Q) = evalE\ \Pi\ (p',\ \sigma',\ e'(\!|lv := I|\!))\ Q$
qed*(G)*

Now, we have finished our proof for the `Putfield` instruction. In total this proof amounts to 1kloc of Isar script and is an example of a non-trivial instantiation proof. Other instructions with similar complexity are `New`, `Invoke`, `Return` and `Throw`. The remaining instructions are straightforward and all their induction cases can be handled automatically. □

B Bibliography

[1] Jissa website, http://www.quiss.org/jissa/, 2000.

[2] Bytecode engineering library by markus dahm; http://bcel.sourceforge.net, 2002.

[3] Proving Theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, pages 227–290. IOS Press, 2003.

[4] VeryPCC project website in Munich, http://isabelle.in.tum.de/verypcc/, 2003.

[5] M. Abadi and K. R. M. Leino. A Logic of Object-Oriented Programs. In *Verification: Theory and Practice*, volume 2772 of *Lect. Notes in Comp. Sci.*, pages 11–41. Springer-Verlag, 2004.

[6] A. W. Appel. Foundational proof-carrying code. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258, June 2001.

[7] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, pages 243–253, January 2000.

[8] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. Technical report, Princeton University, October 2000.

[9] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

[10] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resource verification. In *Proc. 17th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs 2004)*, pages 34–49. Springer Verlag, 2004.

[11] D. Aspinall and M. Hofmann. Another type system for in-place update. In D. L. Metayer, editor, *European Symposium on Programming*, pages 36–52, 2002.

[12] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. (Paperback: 1999).

[13] C. Ballarin. Locales and Locale Expressions in Isabelle/Isar., 2004.

[14] F. Bannwart and P. Müller. A program logic for bytecode. In *Proceedings of the 1st Workshop on Bytecode Semantics, Verification and Transformation, Electronic Notes in Computer Science*, 2005. to appear.

[15] S. Berghofer. *Proofs, Programs and Higher Order Logic.* PhD thesis, Institut für Informatik, Technische Universität München, 2004.

[16] S. Berghofer and T. Nipkow. Proof terms for simply typed higher order logic. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics*, volume 1869 of *Lect. Notes in Comp. Sci.*, pages 38–52. Springer-Verlag, 2000.

[17] S. Berghofer and T. Nipkow. Executing higher order logic. In *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *Lect. Notes in Comp. Sci.*, pages 24–40. Springer-Verlag, 2002.

[18] L. Beringer, M. Hofmann, A. Momigliano, and O. Shkaravska. Automatic certification of heap consumption. In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR2004)*, volume 3452 of *Lect. Notes in Comp. Sci.* Springer Verlag, March 2005.

[19] L. Beringer, K. MacKenzie, and I. Stark. A functional form for imperative mobile code. In *Proceedings of the 2nd EATCS Workshop on Foundations of Global Computing (FGC'03), Electronic Notes in Theoretical Computer Science.*, volume 85(1), June 2003.

[20] A. Biere, A. Cimaati, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. In M. V. Zielkowitz, editor, *Advances in Computers - Highly dependable software*, volume 58, 2003.

[21] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In R. Cytron and R. Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI'03)*, pages 196–207. ACM Press, 2003.

[22] F. D. Boer and C. Pierik. A Syntax-Directed Hoare Logic for Object-Oriented Programming Concepts. In *Proceedings of Formal Methods for Open Object-based Distributed Systems (FMOODS)*, LNCS. Springer, 2003.

[23] P. Boizumault. *The Implementation of Prolog.* Princeton University Press, 1993.

[24] R. Bornat. Proving pointer programs in Hoare Logic. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC 2000)*, volume 1837 of *Lect. Notes in Comp. Sci.*, pages 102–126. Springer-Verlag, 2000.

[25] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: a developer-oriented approach. In G. S. Mandrioli D, Araki K, editor, *Formal Methods: International Symposium of Formal Methods Europe (FME 03)*, volume 2805 of *Lect. Notes in Comp. Sci.*, pages 422–439, 2003.

[26] R. 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.

[27] A. Chaieb. Isabelle trifft Presburger Arithmetik. Master's thesis, Institut für Informatik, TU München, 2003.

[28] A. Chaieb. Proof-producing program analysis. Technical report, TU, München, Dec. 2004.

[29] A. Chaieb and T. Nipkow. Generic Proof synthesis for Presburger Arithmetic. Technical report, TU München, 2003.

[30] B.-Y. E. Chang, A. Chlipala, G. C. Necula, and R. R. Schneck. The Open Verifier Framework for Foundational Verifiers. In *In Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'05)*. ACM SIGPLAN Notices, 2005.

[31] J. Chen, D. Wu, A. W. Appel, and H. Fang. A Provably Sound TAL for Back-end Optimization. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003)*, pages 208–219. ACM, June 2003.

[32] M. Clint and C. A. R. Hoare. Program proving: Jumps and functions. In *Acta informatica*, volume 1. Springer-Verlag, 1972.

[33] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Proc. ACM SIGPLAN conf. Programming Language Design and Implementation (PLDI)*, pages 95–107, 2000.

[34] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.

[35] K. Crary. Toward a foundational typed assembly language. In *Proccedings of the 2003 Symposium on Principles of Programming Languages (POPL'03)*. ACM Press, January 2003.

[36] K. Crary and S. Sarkar. A metalogical approach to foundational certified code. Technical report, CMU Technical Report CMU-CS-03-108, January 2003.

[37] G. Dantzig and B. C. Eaves. Fourier-motzkin elimination and its dual. *J. Combinatorial Theory A*, 14:288–297, 1973.

[38] G. B. Dantzig. *Lineare Programmierung und Erweiterungen.* Springer, 1966.

[39] A. Dehne. Beweiserzeugende Programmanalyse: Intervallanalyse. Master's thesis, Technische Universität München, 2005.

[40] D. L. Detlefs, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical report, Compaq Systems Research Center, 1998.

[41] A. Deutsch. Static verification of dynamic properties. Technical report, PolySpace Technologies, 2003.

[42] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[43] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide version 5.8. Technical Report 154, INRIA, May 1993.

[44] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980.

[45] Fischer and Rabin. Super-exponential complexity of presburger arithmetic. In *SIAMAMS: Complexity of Computation: Proceedings of a Symposiu m in Applied Mathematics of the American Mathematical Society and the Society fo r Industrial and Applied Mathematics*, 1974.

[46] M. Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, 1996.

[47] R. W. Floyd. Assigning meanings to programs. In J. Schwartz, editor, *Proceedings: Symposium on Applied Mathematics*, volume 19, pages 19–32, 1967.

[48] E. Ganser, E. Koutsofios, and S. North. Drawing graphs with *dot.* Technical report, AT&T, 2002. http://www.graphviz.org.

[49] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[50] N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. 17th IEEE Symp. Logic in Computer Science*, pages 89–100, July 2002.

[51] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:567–580,583, 1969.

[52] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *In Proceedings of the 30th Symposium on Principles of Programming Languages (POPL'03)*, pages 185–197. ACM Press, January 2003.

[53] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 2nd edition, 1990.

[54] G. Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[55] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, Mar. 2004. to appear in TOPLAS.

[56] G. Klein and M. Wildmoser. Verified bytecode subroutines. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, volume 2758 of *Lect. Notes in Comp. Sci.*, pages 55–70. Springer Verlag, September 2003.

[57] C. League, Z. Shao, and V. Trifonov. Precision in practice: A type-preserving Java compiler. Technical Report YALEU/DCS/TR-1223, Department of Computer Science, Yale University, Mar. 2002.

[58] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. Jml reference manual (draft). Technical report, 2004.

[59] T. Lev-Ami, T. Reps, M. Sagiv, and T. Wilhelm. Putting static analysis to work for verification: A case study in issta 2000. Technical report, 2000.

[60] L. Mauborgne. Astrèe: Verification of absence of runtime error. In R. Jacquart, editor, *Building the information society (WCC'04)*, pages 385–392. Kluwer, 2004.

[61] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 199:200–227, 2005.

[62] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 6th International Conference, TACAS 2000*, volume 1785 of *Lect. Notes in Comp. Sci.*, pages 63–77, 2000.

[63] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[64] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 85–97. ACM Press, 1998.

[65] M. Müller-Olm and H. Seidl. Computing polynomial program invariants. *Information Processing Letters*, 91(5):233–244, 2004.

[66] G. Necula and S. Rahul. Oracle based checking of untrusted software. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, 2001.

[67] G. C. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[68] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.

[69] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *13th IEEE Symp. Logic in Computer Science (LICS'98)*, pages 93–104. IEEE Computer Society Press, 1998.

[70] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Programming Languages and Systems*, 1(2):245–257, 1979.

[71] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27:356–364, 1980.

[72] H. R. Nielson and F. Nielson. *Semantics with Applications*. Wiley, 1992.

[73] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *Lect. Notes in Comp. Sci.*, pages 180–192. Springer-Verlag, 1996.

[74] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lect. Notes in Comp. Sci.* Springer, 2002.

[75] D. v. Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

[76] M. Pavlova and L. Burdy. Java bytecode specification and verification. In *ACM Symposion on Applied Computing (SAC06)*, 2006. to appear.

[77] F. Pfenning and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Automated Deduction — CADE-16*, volume 1632 of *Lect. Notes in Comp. Sci.*, pages 202–206. Springer-Verlag, 1999.

[78] B. C. Pierce. *Advanced Topics in Types and Programming Languages.* The MIT Press, 2005.

[79] C. Quigley. *A Programming Logic for Java Bytecode Programs.* PhD thesis, University of Glasgow, 2004.

[80] C. Sacerdoti Coen. Tactics in modern proof-assistants: The bad habit of overkilling. In R. J. Boulton and P. B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, number EDI-INF-RR-0046 in Informatics Report Series, pages 352–367. Division of Informatics, University of Edinburgh, Edinburgh, Scotland, UK, September 2001.

[81] D. Sanella and M. Hofmann. Mobile ressource guarantees, eu openfet project, http://www.dcs.ed.ac.uk/home/mrg, 2002.

[82] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 318–329, New York, NY, USA, 2004. ACM Press.

[83] N. Schirmer. *Verification of sequential imperative programs in Isabelle/HOL.* PhD thesis, Institut für Informatik, Technische Universität München.

[84] R. Schneck. *Extensible untrusted code verification.* PhD thesis, University of California, Berkeley, 2004.

[85] S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In C. Hankin, editor, *Proc. of Static Analysis Symposium (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 320–335, London, UK, September 2005. Springer. To appear.

[86] S. Seo, H. Yang, and K. 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.

[87] F. Spoto. JULIA: A Generic Static Analyser for the Java Bytecode. In *Proc. of the 7th Workshop on Formal Techniques for Java-like Programs, FTfJP'2005*, Glasgow, Scotland, July 2005.

[88] Sun Microsystems. *The Java Virtual Machine Specification*, Aug. 1995.

[89] G. Tan, A. W. Appel, K. N. Swadi, and D. Wu. Construction of a semantic model for a typed assembly language. In B. Steffen and G. Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI*

2004, Venice, January 11-13, 2004, Proceedings, volume 2937 of *Lecture Notes in Computer Science*. Springer, 2004.

[90] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In T. Margaria and W. Yi, editors, *TACAS 2001 - Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 113–127, Genova, Italy, Apr. 2001. Springer-Verlag.

[91] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. *Lecture Notes in Computer Science*, 2031:299+, 2001.

[92] M. Wenzel. *The Isabelle/Isar Reference Manual*. TU München, 1999.

[93] M. 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 *Lect. Notes in Comp. Sci.*, pages 167–183. Springer-Verlag, 1999.

[94] M. Wildmoser. Subroutines and java bytecode verification. Master's thesis, Technische Universität München, 2002.

[95] M. Wildmoser, A. Chaieb, and T. 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. to appear.

[96] M. Wildmoser and T. Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind and A. Bunker, editors, *Proc. 17th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'04)*, volume 3223 of *Lect. Notes in Comp. Sci.*, pages 305–320. Springer Verlag, September 2004.

[97] M. Wildmoser and T. Nipkow. Asserting bytecode safety. In M. Sagiv, editor, *Proceedings of the 14th European Symposium on Programming (ESOP 2005)*, volume 3444 of *Lect. Notes in Comp. Sci.*, pages 326–341. Springer Verlag, 2005.

[98] M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In J.-J. Levy, E. W. Mayr, and J. C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd Int. Conf. on Theoretical Computer Science (TCS2004)*, pages 333–347. Kluwer Academic Publishers, August 2004.

[99] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

List of Figures

Index