Institut für Informatik
der Technischen Universität München

Reasoning about Terminating Functional Programs

Konrad Slind

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.

Vorsitzende:		Univ.-Prof. Dr. Angelika Steger
Prüfer der Dissertation:		
	1.	Univ.-Prof. Tobias Nipkow, Ph.D.
	2.	Univ.-Prof. David Basin, Ph.D., Albert-Ludwigs-Universität Freiburg

Die Dissertation wurde am 25.06.99 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 4.11.99 angenommen.

Abstract

This thesis addresses two basic problems with the current crop of mechanized proof systems. The first problem is largely technical: the act of soundly introducing a recursive definition is not as simple and direct as it should be. The second problem is largely social: there is very little code-sharing between theorem prover implementations; as a result, common facilities are typically built anew in each proof system, and the overall progress of the field is thereby hampered.

We use the application domain of functional programming to explore the first problem. We build a pattern-matching style recursive function definition facility, based on mechanically proven wellfounded recursion and induction theorems. Reasoning support is embodied by automatically derived induction theorems, which are customised to the recursion structure of definitions. This provides a powerful, guaranteed sound, definition-and-reasoning facility for functions that strongly resemble programs in languages such as ML or Haskell. We demonstrate this package (called TFL) on several well-known challenge problems.

In spite of its power, the approach suffers from a low level of automation, because a termination relation must be supplied at function definition time. If humans are to be largely relieved of the task of proving termination, it must be possible for the act of defining a recursive function to be completely separate from the act of finding a termination relation for it and proving the ensuing termination conditions. We show how this separation can be achieved, while still preserving soundness. Building on this, we present a new way to define program schemes and prove high-level program transformations.

Since the second problem is largely social, we cannot solve it alone; however, we do present an artifact that marks a path to a brighter future. In particular, we show that the sophisticated algorithms implemented in TFL can be parameterized by a higher order logic proof system. The package has been instantiated to HOL and Isabelle-HOL, two quite different mechanizations of higher order logic. In this exercise, we found that the fully formal approach taken to justifying definitions and deriving induction schemes was fundamental in providing the required combination of portability and soundness.

Contents

Acknowledgements

I would like to thank Tobias Nipkow for helping me to get started in Germany, and for his unstinting efforts to improve the quality of this work. Most of the ideas in this thesis were forged in our weekly meetings. Moreover, Tobias waited patiently for this work to finally be completed.

Bernhard Schätz befriended me in Munich when I was just a trembling puppy of an Ausländer. Ever since, he has been a continual source of good humour, help of all kinds, and general encouragement. Without him, this thesis would never have been submitted.

Olaf Mueller was an excellent officemate; his bright and outgoing personality helped make our time together in the office very pleasant.

John Harrison has been a fount of technical advice and theorem-proving know-how. As well, he has been a regular and entertaining email correspondent.

My proof readers John Harrison, Mark Staples, Michael Norrish, and Joe Hurd are due thanks for their keen-eyed reading of some fairly awful prose. Thanks are also due to Mike Gordon, Richard Boulton, and Alan Bundy for their forbearance while this work was being finished.

The bookends of my working days in Munich were brightened by M. Schmiedchen, B. Schwarz, R. Schneider, H. Grünert, K. Penta, D. Hüdelmeier, and especially Horst Hübsch.

I met Alexandra Karl in Munich; she stuck it out with me through many years of odd hours and disruption, while somehow also achieving her own academic goals. Her love and suppport has enabled us to prevail, and for that, a proper thank-you lies far from the realm of words on a page. But nevermind: thanks Alex!

Chapter 1

Introduction

A *program* is a text (a piece of syntax, marks on a page, ...) that may be regarded as a sequence of machine operations: programs bring computers to life. Typically, a program has been created by a human and is intended to achieve certain goals of its author(s). However, it is often not clear whether or not a program fulfills its author's expectations; in fact it seems that programs rarely perform exactly as expected. Further, most authors do not (or can not) express their expectations precisely enough to ascertain whether or not their program works properly. This state of affairs is the *raison d'être* of the field of software engineering, which studies principles for producing and maintaining 'good' software. The techniques studied in software engineering cover a vast range and we shall only discuss a few:

Lifecycle methods. In these [14], a model of the lifecycle of software is used as a basis for a methodology governing the way software is produced and maintained. Advocates of a given style will assert that better software results if the methodology is followed. Such methods are often used on large projects and have been found to be beneficial.

Advanced programming languages. Programming languages having exotic type systems, parameterized modules, and incorporating object-oriented facilities, concurrency, and distribution are under intensive investigation[1, 10, 97]. Of course, the intent is that, with such languages, it should be easier to write and maintain sophisticated programs than currently possible.

Formal methods. This approach is inspired by mathematical logic: the idea is to represent (some aspect of) programs in a formal language, *i.e.*, a language with mathematically described syntax and semantics. In this setting, a program is no longer just a text but also a mathematical object; precise properties of programs can be stated and established. In particular, *correctness*, *i.e.*, that a program performs as expected, can be rigorously shown.

Of course, these techniques can overlap, and there have been fruitful combinations of them. For example, lifecycle methods can benefit from incorporating formal descriptions: many incoherencies can be caught early in the lifecycle if requirements specifications are written in a formal language. The rest of the development may proceed non-formally but a large and expensive class of errors is detected early, with corresponding large savings.

The work described in this thesis is a contribution to formal methods. The main goal of formal methods is the production of 'error-free software' (or hardware). In contrast to a lifecycle method, where one puts one's faith in the methodology to weed out errors, and thus it is possible that some errors slip through, a formal method is supposed to provide the highest degree of precision: complete assurance that no logical errors exist in the software or hardware under consideration.

The field of formal methods is undergoing rapid growth as it tries to find its proper niche within software engineering and other fields such as mathematics. A large number of formal methods have been proposed. In general, these reflect the nature of the applications (concurrency, distributed systems, hybrid systems, imperative sequential programs, functional programs, hardware, *etc.*), the various formalisms suitable for modelling computational phenomena (propositional logic, temporal logic, first order logic, higher order logic, set theory, category theory, automata theory, process algebra, *etc.*), and the level of automation that a formalism provides (the weaker the expressive power of a formalism, the more automatable it is).

Functional programming

Our specific interest is in proving the correctness of pure functional programs. Functional programming has its roots in the seminal logical research of the 1920s and 1930s. Schönfinkel and later Curry discovered that all functions could be reduced to functions having a single arity, thus starting the study of *combinators* which occupied Curry for many years thereafter [28] and which have since become an important technique in the implementation of functional languages. Church initially put forth his *lambda calculus* as a foundation of mathematics. It was shown to be inconsistent, but later on Church used a typed version of it as the basis for a higher order logic [23]. The untyped lambda calculus proved to be of equal strength to Turing machines; however, for many years the lambda calculus languished until the advent of the computer. McCarthy used some aspects of the lambda notation in his LISP language [70]. In the 1960s, Landin used the lambda calculus as a notation that captured aspects common to a wide number of programming languages [62]. He also invented the SECD machine, an important technique in implementing functional languages [61]. Burstall introduced constructor-based datatypes, 'case' statements and structural induction in his seminal paper of 1969 [21]. In spite of its evident utility in capturing important aspects of programming languages, there was some suspicion among logicians about the mathematical meaning of terms in the untyped calculus. A mathematical semantics for programs was finally achieved by Dana Scott in the late 1960s with his theory of domains [94]. In the middle of the 1970s, Robin Milner and his colleagues presented LCF [42], a proof system for Scott's logic. This system was coded in ML, an influential implementation of the typed lambda calculus.

ML marks the beginning of a new era in functional programming, combining functions as first class citizens, type inference, and pattern matching. ML is not purely functional, since its design included reference cells, which allow imperative operations. Some purely functional languages, which disallow any such manipulations of a hidden state also sprang up around this time [100]. Since these beginnings, functional programming and the lambda calculus have been the subject of a vast amount of research, which we cannot hope to do justice to here.

Instead we will discuss some motivations for this interest. A *pure* functional program may be understood as a lambda calculus term with some extra syntax added for readability. Therefore, theoretical properties enjoyed by the lambda calculus are inherited by functional programs. Two practical benefits of this have repeatedly been put forward by functional programming advocates: parallelism and correctness. Possibilities for parallelism arise since the lambda calculus has the Church-Rosser property - any terminating evaluation strategy for lambda terms will give the same answer as any other, so a parallel evaluation of a function will give the same answer, hopefully faster, as a sequential evaluation. Although parallel implementations of functional languages have demonstrated impressive performance [13], they are still mainly research prototypes.

1.1 Verification of functional programs

Another strength of functional programming is supposed to be that one can more easily prove correctness. In contrast to an imperative program, one can directly reason using the mathematical meaning of a functional program. This semantic *clarity* is quite appealing.

The LCF system directly implemented Scott's logic, and some interesting correctness proofs were performed in it [25, 83, 84]. The Boyer-Moore theorem prover, known as *Nqthm*, appeared at approximately the same time as LCF, but was quite different in its logical basis and implementation [19]. The principal logical choice made by Boyer and Moore was to represent programs, not as operating over domains, as in LCF, but as total functions. This choice means that the set of programs representable in Nqthm is not Turing complete, as it is in LCF, but is instead restricted to those functions that can be proved to terminate for all inputs. However, this is still a quite interesting set, as it contains many important algorithms. Furthermore, the argument of semantic clarity has even more force in the case of Nqthm, since programs are identified with total functions. The principal benefit of this is that one can freely mix standard mathematics with programs. However, from the vantage point of directly modelling higher order functional programs, the first order term language of Nqthm is inadequate for expressing many interesting algorithms. The current crop of higher order logic proof systems—many of which are descendants of LCF—would seem to be a better fit.

1.1.1 Programs and functions

We have already seen two examples of how programs can be represented in proof systems: the computable functions of LCF, and the total functions of Nqthm. Another possibility would be to define the abstract syntax of the programming language of interest in the logic, and define a notion of program evaluation in terms of these syntax trees. Reasoning about programs would then be carried out by reasoning about evaluation. Such an approach is often called a *deep embedding*. We do not pursue deep embedding in this dissertation since we are more interested in what can be done with the *native* functions of a logic.

Another possibility, which is philosophically very interesting, is to not write programs at all: in a constructive logic, proofs that specifications are satisfiable can be translated into programs that meet those specifications [68, 26]. In principle, one simply proves the satisfiability of a specification and, from that proof, a correct program can be automatically extracted. We shall not pursue this approach either, since it lacks directness: our belief is that it is easier to write programs than to do proofs.

Therefore, we want to directly express programs as the native functions of a higher order logic. By doing so, we are intentionally confusing programs and logical functions, just as in Nqthm. Of course, there are far more functions than programs, so the identification is not accurate. However, we believe that the convenience of working with pure total functions more than balances any worries about whether, *e.g.*, the function one has just proved a property of is computable.

How does a program actually make its way into the logic? For example, we can't allow just any program to be turned directly into a logical function. The perfectly acceptable, although useless, program

$$f(x) = f(x) + 1$$

would, if accepted as a logical function, have the almost immediate consequence that $0 = 1$, which renders the logic useless. A standard (and very general) remedy for such problems is to define total recursive functions by *wellfounded recursion*. A mathematical relation is *wellfounded* if it admits no infinite decreasing chains. From the perspective of programs, if every sequence of recursive calls a program makes can be fit into a wellfounded relation, that function is total, *i.e.*, it terminates. Then the recursion equations defining the function may be validly used inside the logic, *i.e.*, the program becomes a respectable citizen of the logic, and trustworthy proofs of its correctness can be performed.

The topic of this dissertation has now been motivated: we are going to investigate the automation and application of the wellfounded recursion theorem in higher order logic.

1.2 TFL

Our work has been implemented in a system called TFL. The TFL system has the following significant features:

Higher order. Programs are represented by the native functions of higher order logic. This *lightweight* representation means that the proof rules already provided in the logic (*e.g.*, β-conversion, extensionality) can be immediately applied to programs. Since the native functions can be higher order and polymorphically typed, programs can also be higher order and polymorphic, and therefore a wide class of programs from popular functional languages like ML and Haskell can be directly formalized and reasoned about.

Fully formal. Logically, the environment is a *conservative extension.* The machinery of TFL uses inference steps to instantiate and manipulate the wellfounded recursion theorem. Therefore, using it will not suddenly allow false theorems to be derived. In particular, definitions of recursive programs can be made without any worries about introducing inconsistency.

Portable. As a result of its fully formal foundation, TFL is portable in the following sense: it can be instantiated to different proof systems. As part of our work, we have instantiated TFL to the (quite different) Isabelle/HOL and hol90 mechanized theorem provers.

Automatic extraction of termination conditions. We make novel use of congruence based contextual rewriting to extract *termination conditions* from a proposed recursive definition.

Pattern matching and variable binding operators. Our syntax effectively deals with the pattern-matching style of definition popular in functional programming languages. Another feature of functional languages is *let* bindings. These are used to share computation, and therefore are of great practical utility in writing efficient algorithms. The termination condition extractor handles such variable-binding constructs. For example, quicksort can be defined in TFL by the following programs, which builds the inputs for the two recursive calls using a single pass:

$$\begin{array}{rcl}
\mathsf{part}(P,[\,],l_1,l_2) & \equiv & (l_1,l_2) \\
\mathsf{part}(P,h::t,l_1,l_2) & \equiv & \texttt{if}\ P\ h\ \ \texttt{then}\ \mathsf{part}(P,t,h::l_1,l_2) \\
 & & \qquad\quad\ \ \texttt{else}\ \mathsf{part}(P,t,l_1,h::l_2)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{fqsort}(ord,[\,]) & \equiv & [\,] \\
\mathsf{fqsort}(ord,h::t) & \equiv &
\end{array}$$

$$\begin{array}{l}
\quad\mathsf{let}\ (l_1,l_2) = \mathsf{part}(\boldsymbol{\lambda} y.\ ord\ y\ h, t, [\,], [\,]) \\
\quad\mathsf{in} \\
\qquad \mathsf{fqsort}(ord,l_1)\ @\ [h]\ @\ \mathsf{fqsort}(ord,l_2).
\end{array}$$

Deferral of termination arguments. In other systems, termination of a function needs to be proved before it can be used. This can be a bother, especially since it seems to make good software engineering sense to be able to define a *collection* of functions separately from proving their termination. The result is that defining functions is very easy in TFL, and the potentially difficult work of finding a wellfounded relation under which a function terminates can be postponed, although never avoided.

Induction always. For each recursive function definition, TFL derives a customized principle of induction, by instantiating and manipulating the wellfounded induction theorem. In our induction theorems, the property to be proved is assumed to hold for arguments to recursive calls, and the task is to show that the property holds for the original call. If that can be shown, then the property holds for every invocation of the function.

For instance, if we describe Euclid's algorithm (on the natural numbers) with the following equations:

$$\begin{array}{l}
\mathsf{gcd}(0,y) \equiv y \\
\mathsf{gcd}(\mathsf{Suc}\ x, 0) \equiv \mathsf{Suc}\ x \\
\mathsf{gcd}(\mathsf{Suc}\ x, \mathsf{Suc}\ y) \\
\qquad \equiv \texttt{if}\ y \leq x\ \texttt{then}\ \mathsf{gcd}(x-y,\ \mathsf{Suc}\ y)\ \texttt{else}\ \mathsf{gcd}(\mathsf{Suc}\ x,\ y-x)
\end{array}$$

we get the following induction principle for reasoning about gcd:

$$\begin{array}{l}
\forall P.\ \ (\forall y.P(0,y))\ \wedge \\
\qquad (\forall x.P(\mathsf{Suc}\ x, 0))\ \wedge \\
\qquad (\forall x\ y.(\neg(y \leq x) \supset P(\mathsf{Suc}\ x, y-x))\ \wedge \\
\qquad\qquad\qquad (y \leq x\ \ \supset P(x-y, \mathsf{Suc}\ y)) \supset P(\mathsf{Suc}\ x, \mathsf{Suc}\ y)) \\
\qquad \supset \forall v\ v_1.\ P\ (v, v_1).
\end{array}$$

TFL proves the *pattern completeness* property as part of deriving the induction theorem, helping to ensure its validity.

1.3 Contributions

The following are the new contributions made by our work:

- TFL is, to our knowledge, the first example of a sophisticated proof tool that soundly operates in more than one proof system.

- We demonstrate that a definition mechanism for terminating general recursive functions can be implemented purely by deduction, *i.e.*, it doesn't have to be a hardwired mechanism, as it is in every other mechanized proof system we know of.

- Our method for extracting termination conditions is unique in that it operates purely by inference and is more flexible than other schemes. This is due to the novel choice to implement it via contextual rewriting.

- We show that *program schemes* may be defined more easily and more abstractly than allowed in previous mechanizations. These can be used to prove *formal* program transformations which are also simpler and more abstract than previous attempts in mechanized systems.

- Nested recursive programs are notoriously difficult to deal with in a logic of total functions. Some authors have asserted that partial and total correctness need to be proved *simultaneously* for such programs. We show that this is not true:[1] a nested recursive program is just like any other, although nestedness can make its termination proof more intricate. We give new and simpler correctness proofs for some well-known nested functions.

1.4 Related research

We shall give a cursory overview of related systems, concentrating on their ability to represent and reason about functional programs. Such an analysis often goes to the heart of such systems, since functions are such a pervasive notion in mathematics.

ACL2 The Nqthm system has evolved into *ACL2*, which adds many features but seems to leave the underlying logic essentially unchanged. The function definition principle of ACL2 is based on wellfounded recursion. However, the wellfounded recursion theorem itself is not available to the user other than through the principle of definition. When given a function to define, ACL2 computes termination conditions for the function and attempts to prove them automatically, using a hard-wired termination relation and previously proved *termination lemmas*. If that proof fails, the user must state

[1] This observation has also been (independently) made by Juergen Giesl [40].

and prove the required termination lemma before the function can be defined. In the case of nested recursions, the ACL2 user must prove that the nested description is equivalent to an already defined function. For each defined function, ACL2 builds an induction scheme, which is again an internal object that is not (naturally) available for the user to manipulate. The great strength of the ACL2 system is in its automation: it automatically applies a number of powerful heuristics, including inducton, in attempting proofs.

ACL2 by default uses a size measure on data as a termination relation; if necessary, the ordinals 'up to ε_0' can be specified. These ordinals are commonly used to do the work of lexicographic combinations of measure functions.

ACL2 does not provide mutual recursion; Boyer and Moore recommend directly defining the so-called 'union' function.

PVS The PVS system [79] is based on a variant of Church's higher order logic with a powerful type system that includes subtypes and dependent types. PVS has been used with great success on many verification problems. Like ACL2, PVS defines functions by implicit appeal to the wellfounded recursion theorem. The relations that may be supplied are measure functions over the arguments of the recursive function, and again, the range of the measure may be the ordinals up to ε_0. The powerful type system of PVS allows nested recursions to be defined by using the type system to require that the nested applications of the function being defined meet the behavioural specification of the function. Currently, PVS does not seem to support mutual recursive definitions.

HOL The HOL system implements a slight generalization of Church's logic in which Church's meta-linguistic use of type variables has been replaced by type variables in the logic. Function definitions in HOL have typically been based on (higher order) primitive recursion. However, there has been some past work on wellfounded recursion in the HOL system by vanderVoort [102]. His work attempted to reduce non-primitive recursions to primitive recursions. Agerholm has done some interesting work which combines a simplified domain-theory with termination proofs, resulting in a flexible system for total function definition[3]. Mutual primitive recursion is supported in HOL by a package for mutually recursive datatypes.

Isabelle The Isabelle/HOL instantiation has provided a wellfounded recursion operator and a wellfounded recursion theorem for some time. This was freely available for use, however, such definitions tended to be quite low level and suffered from a lack of automation: the intended recursion equations had to be derived, and the induction theorem would also have to be derived by hand. Higher order primitive recursion, including mutual recursion, is supported by a datatype package. An extension of Isabelle/HOL to semantically embed LCF yielded HOLCF[89], in which functions may be defined via a domain theoretic fixpoint operator. This has the nice property that termination proofs need never be done; however, the user pays the price of having two somewhat incompatible notions of function space in HOLCF. Müller [76] worked out a methodology for dealing effectively with the hybrid setting, using HOLCF to handle thorny modelling problems, while conducting the bulk of the proof activity with the total functions of HOL.

LAMBDA The LAMBDA system essentially implements the HOL logic, and justifies recursive definitions by means of a least fixedpoint construction. It automatically synthesizes termination constraints when a function is defined [36]. Mutual and nested recursion is allowed. However, a direct connection with termination does not seem to be included: in LAMBDA, the intent is that once the 'termination conditions' are eliminated from the assumptions of the recursion equations, the function has been proved to terminate. However, the system does not require that the relation computed from the structure of the recursion (which is used to phrase the 'termination conditions') be wellfounded. As a result, the semantics of certain recursive definitions in LAMBDA must be carefully argued.

Although induction theorems are not automatically generated for function definitions in LAMBDA, Holger Busch has made an extensive investigation into wellfounded induction in the LAMBDA system [47].

Coq The Coq system implements the Calculus of Inductive Constructions, a powerful constructive type theory. The usual activity in Coq is to extract programs from proofs. However, the thesis of Parent [80] investigated a way of partially synthesizing a proof given a program and its specification. The thesis of Cornes [27] featured a rich pattern language allowing overlapping patterns, so-called 'as' patterns and pattern matching on dependent types [66]. Another constructive type theory implementation that supports pattern matching in definitions is Alf [64].

CYNTHIA is a system for editing ML programs [107]. The system uses the proofs-as-programs interpretation from constructive logic to offer a smart editor. The available editing operations are such that several weak forms of correctness are maintained as invariants: pattern completeness, termination, and well-typedness. The user works by altering an existing correct program. Some of the editing operations are: to change the type of the program, to change the recursion structure of the program, to change the pattern-matching structure, and to rename variables. Each editing step leads to proofs that show that the invariants are maintained. Walther's approach [104] is used to automate termination proofs, which can arise when the recursion structure is changed. Whittle has tested his system out on novice ML programmers and found that they learned better. Perhaps the most impressive aspect of CYNTHIA is the complete hiding of a formal method under a suitable interface.

1.5 Organization

In Chapter 2 we discuss the underlying logical and mathematical basis of the system. In Chapter 3, the details of mechanizing the process of recursive definition and the production of induction theorems are given. In Chapter 4, we will describe the challenges raised by nested recursion, and show how the approach in Chapter 3 needs to be modified to accommodate nestedness. In Chapter 5, we present a number of examples that illustrate our method. We conclude with Chapter 6. In the Appendix, we discuss the system architecture.

Chapter 2

Logical Basis

This chapter provides an overview of the logical base required by TFL. We begin by describing the version of higher order logic employed, as well as associated notation, and recall some basic facts about datatypes. After that, we show how to construct the main logical tools used our work: wellfoundedness, induction, and recursion. Subsequently, support for termination proofs in the form of a small collection of wellfoundedness theorems is discussed. The chapter finishes with a description of an implementation of contextual rewriting, which is an essential ingredient in TFL.

This chapter is intended to provide an accessible description of the requirements of TFL, so it will have a tutorial flavour at times.

2.1 Higher order logic

We choose to work inside a typed higher order predicate calculus [43], derived from Church's Simple Theory of Types [23]. The name commonly used for this logic is HOL. The HOL logic is classical and has a set theoretic semantics, in which types denote non-empty sets and the function space denotes total functions. Several mature computer mechanizations of this logic exist [86, 7, 50], making it—in our opinion—the best available candidate logic to code a portable proof tool for. Porting TFL to other logical systems of adequate expressiveness, such as set theory, is an intriguing possibility that we shall not address in this dissertation.

The HOL logic is built on the syntax of a lambda calculus with an ML-style polymorphic type system. In the following, we define types and terms as a prelude to discussing some aspects of the required rules of inference, axioms, and principles of definition. In general, the discussion will neglect semantic matters, since we are mainly interested in mechanizing proofs inside the deductive system.

The syntax of the logic is based on signatures for types (Ω) and terms (Σ_Ω). The type signature assigns arities to type operators, while the term signature delivers the types of constants.

Definition 1 (HOL types) *The set of HOL types is the least set closed under the following rules:*

type variable. *There is a countable set of type variables, which are represented with Greek letters, e.g., α, β, etc.*

compound type. *If c in Ω has arity n, and each of $ty_1, \ldots ty_n$ is a type, then $c(ty_1, \ldots ty_n)$ is a type.*

□

A type constant is represented by a 0-ary compound type. A large collection of types can be definitionally constructed in HOL, building on the initial types found in Ω: truth values (bool), function space (written $\alpha \rightarrow \beta$), and an infinite set of individuals (ind).

Terms are typed $\boldsymbol{\lambda}$-calculus expression built with respect to Σ_Ω. When we wish to show that a term M has type τ, the notation $M : \tau$ is used. Note in the following that, in building a term, each subterm is assigned a unique type.

Definition 2 (HOL terms) *The set of terms is the least set closed under the following rules:*

Variable. *if v is a string and ty is a type built from Ω then $v : ty$ is a term.*

Constant. *$c : ty$ is a term if $c : \tau$ is in Σ_Ω and ty is an instance of τ, i.e., there exists a substitution for type variables θ, such that each element of the range of θ is a type in Ω and $\theta(\tau) = ty$.*

Combination. *$(M\ N)$ is a term of type β if M is a term of type $\alpha \rightarrow \beta$ and N is a term of type α.*

Abstraction. *$\boldsymbol{\lambda} v.\ M$ is a term of type $\alpha \rightarrow \beta$ if v is a variable of type α and M is a term of type β.*

□

Initially, Σ_Ω contains constants denoting equality ($=$), implication ($\supset$), and a description operator (ε, which we will discuss more thoroughly in the sequel). Types and terms form the basis of the *prelogic*, in which basic algorithmic manipulations on types and terms are defined: the variables of a type, the free variables in a term, α-convertibility, substitution inside types and terms, instantiation of type variables in terms, and β-conversion. These notions are standard, although correct and efficient implementations can be difficult to achieve, and we simply assume their existence.

For describing substitution, the notation $[M_1 \mapsto M_2]\,N$ is used to represent the term N where all free occurrences of M_1 have been replaced by M_2. Of course, M_1 and M_2 must have the same type in this operation. During substitution, every binding occurrence of a variable in N that would capture a free variable in M_2 is renamed to avoid the capture taking place.[1]

2.1.1 Deductive system

For us, the exact collection of primitive inference rules and axioms is not important, provided all the usual classical rules for quantifiers and connectives become available. In Figure 2.1, a useful set of inference rules is outlined, along with the axioms of the HOL logic. The derivable theorems are just those that can be generated by using the axioms and inference rules of Figure 2.1. For a more parsimonious presentation of this deductive system see [43] or Appendix A of [51].

A theorem with hypotheses $P_1, \ldots, P_k$ and conclusion Q (all of type bool) is written

$$[P_1, \ldots, P_k] \vdash Q.$$

In the presentation of some rules, *e.g.*, $\vee$-elim, the following idiom is used: $\Gamma, P \vdash Q$. This denotes a theorem where P occurs as a hypothesis. A later reference to Γ then actually means $\Gamma - \{P\}$, *i.e.*, if P had already been among the elements of Γ, it would now be removed.

Some rules, noted by use of the asterisk in Figure 2.1, have restrictions on their use or require special comment:

- $\forall$-intro. The rule application fails if x occurs free in Γ.
- $\exists$-intro. The rule application fails if N does not occur free in P. Moreover, only *some* designated occurrences of N need be replaced by x. The details of how these occurrences are singled out vary from system to system.
- $\exists$-elim. The rule application fails if the variable v occurs in $\Gamma \cup \Delta \cup \{P, Q\}$.
- Abs. The rule application fails if v occurs free in Γ.
- tyInst. A substitution θ mapping type variables to types is applied to each hypothesis, and also to the conclusion.

There is some variance among mechanized proof systems in how rules of inference are implemented. Overcoming this was an important part of our work. We shall defer a detailed discussion of this point to Appendix A.2.2.

One essential requirement of our work is $\varepsilon : (\alpha \to \mathsf{bool}) \to \alpha$, Hilbert's *indefinite* description operator. A description term $\varepsilon x : \tau.\ P\ x$ is interpreted as

[1] In some implementation techniques for the lambda calculus, *e.g.*, deBruijn terms [29], capture cannot happen, therefore renaming can be dispensed with. Note, however, that in interactive theorem provers it is common for deBruijn terms to undergo renaming when terms are printed out; otherwise, extremely confusing and unhelpful syntax may be displayed.

$\supset$ -intro	$\dfrac{\Gamma \vdash Q}{\Gamma - \{P\} \vdash P \supset Q}$	$\dfrac{\Gamma \vdash P \supset Q \quad \Delta \vdash P}{\Gamma \cup \Delta \vdash Q}$	$\supset$ -elim
$\wedge$-intro	$\dfrac{\Gamma \vdash P \quad \Delta \vdash Q}{\Gamma \cup \Delta \vdash P \wedge Q}$	$\dfrac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P \quad \Gamma \vdash Q}$	$\wedge$-elim
$\vee$-intro	$\dfrac{\Gamma \vdash P}{\Gamma \vdash P \vee Q,\ \Gamma \vdash Q \vee P}$	$\dfrac{\begin{array}{c}\Gamma_1 \vdash P \vee Q \\ \Gamma_2, P \vdash M \quad \Gamma_3, Q \vdash M\end{array}}{\Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \vdash M}$	$\vee$-elim
$\forall$-intro*	$\dfrac{\Gamma \vdash P}{\Gamma \vdash \forall x.\ P}$	$\dfrac{\Gamma \vdash \forall x.\ P}{\Gamma \vdash [x \mapsto N]P}$	$\forall$-elim
$\exists$-intro*	$\dfrac{\Gamma \vdash P}{\Gamma \vdash \exists x.\ [N \mapsto x]P}$	$\dfrac{\begin{array}{c}\Gamma \vdash \exists x.\ P \\ \Delta, [x \mapsto v]P \vdash Q\end{array}}{\Gamma \cup \Delta \vdash Q}$	$\exists$-elim*
Assume	$P \vdash P$	$\vdash M = M$	Refl
Sym	$\dfrac{\Gamma \vdash M = N}{\Gamma \vdash N = M}$	$\dfrac{\Gamma \vdash M = N,\ \Delta \vdash N = P}{\Gamma \cup \Delta \vdash M = P}$	Trans
Comb	$\dfrac{\Gamma \vdash M = N,\ \Delta \vdash P = Q}{\Gamma \cup \Delta \vdash M\ P = N\ Q}$	$\dfrac{\Gamma \vdash M = N}{\Gamma \vdash (\boldsymbol{\lambda} v.M) = (\boldsymbol{\lambda} v.N)}$	Abs*
tyInst*	$\dfrac{\Gamma \vdash M}{\theta(\Gamma) \vdash \theta(M)}$	$\vdash (\boldsymbol{\lambda} v.M)N = [v \mapsto N]M$	β-conv

Bool	$\vdash P \vee \neg P$
Eta	$\vdash (\boldsymbol{\lambda} v.\ M\ v) = M$
Select	$\vdash P\ x\ \supset P(\varepsilon x.\ P\ x)$
ImpEq	$\vdash (M \supset N) \supset (N \supset M) \supset M = N$
Infinity	$\vdash \exists f\!:\!\mathsf{ind} \to \mathsf{ind}.\ (\forall x\, y.\ (f\ x = f\ y) \supset (x = y)) \wedge \neg(\forall y. \exists x.\ y = f\ x)$

Figure 2.1: HOL deductive system

follows: it delivers an arbitrary element e of type τ such that $P\ e$ holds. If there is no object that P holds of, then $\varepsilon x : \tau.\ P\ x$ denotes an arbitrary element of τ. This is summarized in the axiom $\vdash \forall P\ x.\ P\ x\ \supset P(\varepsilon x.\ P\ x)$, repeated for emphasis from Figure 2.1.

2.1.2 Definition principles

In large verifications, many axioms are needed to build the layers of formalization plus the extensive libraries of support mathematics that are often required. However, experience has shown that humans are in general quite bad at asserting consistent axioms. One of the most influential methodological developments in verification has therefore been the adoption of *principles of definition* as logical prophylaxis. A definition principle allows the introduction of a new constant into the signature as well as asserting an axiom characterizing it. The essential properties that such a principle should enjoy are that no inconsistency be introduced and that meaning is preserved. By the latter we mean that a definition does not reveal new information about already existing constants; technically, if M is a term not containing an occurrence of defined constant c, then $\vdash M$ is derivable before the definition of c if and only if $\vdash M$ is derivable after the definition of c.

Principle of Definition 1 (Abbreviation-HOL) *Given terms $x\!:\!\tau$ and $M\!:\!\tau$ in signature Σ_Ω, check that*

1. *x is a variable and the name of x is not the name of a constant in Σ_Ω;*

2. *τ is a type in Ω;*

3. *M is a term in Σ_Ω with no free variables; and*

4. *Every type variable occurring in M occurs in τ.*

If all these checks are passed, add a constant $\mathsf{x} : \tau$ to Σ_Ω and introduce a theorem $\vdash \mathsf{x} = M$.
□

Thus invocation of the principle of definition, for suitable x and M, introduces x as an abbreviation for M. It is shown in [43] to be a sound means of extending the HOL logic. A similar principle, used in Isabelle, demands that the constant to be defined already exists in the signature:

Principle of Definition 2 (Abbreviation-Isabelle) *Given terms $\mathsf{c} : \tau$ and $M : \tau$ and signature Σ_Ω, check that*

1. *$\mathsf{c} : \tau$ is a constant in Σ_Ω;*

2. M is a term in Σ_Ω with no free variables and no occurrences of c*;*

3. Every type variable occurring in M occurs in τ; and

4. No axiom $\mathsf{c} = N$ is in Σ_Ω;

If all these checks are passed, introduce the theorem $\vdash \mathsf{c} = M$.

□

In spite of their simplicity, either of these principles of definition is sufficient to base elaborate formal developments on. In fact, one focus in implementations of the HOL logic has been the provision of *derived* principles of definition, which typically use general theorems and inference to reduce complex definition principles such as primitive recursion, recursive datatypes, and inductive definitions to application of the basic principles of definition. This many-to-one approach allows different principles of definition to be provided without requiring changes to the logic or its kernel implementation. Part of our work can be seen as falling into this reductive approach.

For our work, a major virtue of building on such a basic definition principle is that it improves portability: in order to be as portable as possible, the requirements should be as weak as possible. Since TFL imposes only this very simple requirement on the notion of definition, TFL becomes correspondingly more portable.

Finally, we note that the HOL logic also provides principles of *constant specification* and also *type* definition. Our framework does not make direct use of these principles, so we shall refrain from discussing them. However, our framework is dependent on the consequences arising from logical *datatype* definitions (Section 2.2). Since these are derived from the basic principle of type definition we are indirectly using the primitive principle of type definition.

2.1.3 Notation and basic definitions

With the exception of function types (written $\alpha \to \beta$), product types (written $\alpha \# \beta$), and sum types (written $\alpha + \beta$), compound types will be written postfix, *e.g.*, α list. In parsing types, earlier members of the following list have stronger binding power than later members:

$$\#, +, \to .$$

We will use standard notation for the logical operators and quantifiers. In parsing logical expressions, earlier members of the following list of infixes have stronger binding power than later members:

$$=, \wedge, \vee, \supset, \equiv .$$

All infixes associate to the right: thus, $x \land y \land z$ is mapped to the same term as $x \land (y \land z)$. The two forms of equality, $=$ and $\equiv$ denote the same predicate; we often use $\equiv$ as notation to show that a definition is being made. $\overline{x}$ denotes a finite sequence of distinct syntactic objects. A sequence of length n may be described by $\overline{x}_n$. $\forall(M)$ denotes the universal quantification of all free variables in M.

The *context* notation will also be used: the syntax $M[N]$ stands for the term M where all free occurrences of N have been 'marked'; then $M[P]$ means that each marked occurrence of N in M has been replaced by P, also in such a way that the free variables of P do not become bound.

We will sometimes make use of the fact that predicates can be taken as sets in our logical setting. For example, $S_1 \subseteq S2$ may be written as $\forall x.\ S_1\ x \supset S_2\ x$, and *vice versa.*

Function composition is written infix: $f \circ g \equiv \boldsymbol{\lambda} x.f(g\ x)$.

Definition 3 (Let) $\mathsf{Let}\ f\ M \equiv f\ M.$

The '`let`' construct is defined as a higher order function. The concrete syntax `let` $x = M$ `in` N stands for the term $\mathsf{Let}\ (\boldsymbol{\lambda} x.\ N)\ M$. Notice that in the HOL logic—in contrast to ML—polymorphism is not introduced via `let`; rather, it arises from definitions.

The symbols False and True are the two constants (of type bool) denoting truth values in the logic.

Definition 4 (Arb) $\mathsf{Arb} \equiv \varepsilon z : \alpha.\mathsf{True}$

We sometimes have to deal with, or somehow take steps to avoid, the troublesome issue of partial functions. In a logic of total functions, this is a delicate subject, which can sometimes be dealt with by use of an arbitrary but fixed value. The definition of Arb uses the Hilbert choice operator to denote such an element, for each type τ. Arb is fixed because True has no free variables; it is arbitrary because $\boldsymbol{\lambda} v.\mathsf{True}$ holds for every element of τ. Because of the semantics of ε, the predicate $\boldsymbol{\lambda} v.\mathsf{False}$ would serve just as well.

Notation for proofs

To explain algorithms that perform deduction, we will quite often present the proof steps as they unfold, *i.e.*, in the so-called *forward* style. Such explanations will describe a sequence of theorems, and discuss how later theorems in the sequence are derived from earlier ones.

However, at times we will discuss proofs in which *problem decomposition* is used; in such cases the *backward*, or *goal-oriented* style is used. In this style, an initial goal is broken into subgoals until only trivial ones remain. The syntax for a stage of a typical backwards proof will consist of the goal, plus whatever hypotheses are currently in force. Hypotheses will be numbered, and those that have been used, or are not of current interest may be omitted. For example, suppose the transitivity of equality is to be shown. Then the initial goal is represented as follows:

$$x = y \land y = z \supset x = z.$$

Breaking this goal down with some decomposition steps gives the following goal where the hypotheses are written below the line:

$$\begin{array}{rl} \multicolumn{2}{l}{x = z} \\ \hline 0. & x = y \\ 1. & y = z \end{array}$$

Now we may use hypothesis 0 to modify the goal; subsequently, it might disappear from the displayed hypotheses, but note that no renumbering would occur.

$$\begin{array}{rl} \multicolumn{2}{l}{y = z} \\ \hline 1. & y = z \end{array}$$

2.2 Datatypes

A common way of raising the level of abstraction in a logical formalization is to use *datatypes*. This is largely inspired by the success of datatypes in ML, which is due to the fact that a datatype splits into disjoint pieces, so that functions over the datatype can be written by case analysis; consequently, proofs about such functions also proceed by case analysis. A logical datatype $(\alpha_1, \ldots, \alpha_m)\ ty$ declared as

$$(\alpha_1, \ldots, \alpha_m)\ ty \equiv \mathsf{C}_1\ ty_{11}\ \ldots\ ty_{1k_1} \mid \ldots \mid \mathsf{C}_n\ ty_{n1}\ \ldots\ ty_{nk_n}$$

(where all the type variables in $ty_{11} \ldots ty_{nk_n}$ are in $\{\alpha_1, \ldots, \alpha_m\}$), denotes the set of all values that can be finitely built up by application of the constructors $\mathsf{C}_1, \ldots, \mathsf{C}_n$. Constructors are injective, and applications of different constructors always yield different values. The type is recursive if any ty_{ij} in the type declaration is $(\alpha_1, \ldots, \alpha_m)\ ty$. For a datatype specification that admits a solution in HOL[2], a characterizing theorem of the following form can be derived [71]:

[2] The syntax admits the description of datatypes that cannot exist in the HOL logic for cardinality reasons, *e.g.*, the following specification of lambda calculus terms:

$$ty \equiv \mathsf{Var\ num} \mid \mathsf{App}\ ty\ ty \mid \mathsf{Abs}\ (ty \to ty).$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\forall f_1 \ldots f_n.}\\
\multicolumn{3}{l}{\quad \exists! \mathcal{H} : (\alpha_1, \ldots, \alpha_m)\ ty \to \beta.}\\
\quad\quad \forall x_{11} \ldots x_{1k_1}.\ \mathcal{H}(\mathsf{C}_1\ x_{11} \ldots x_{1k_1}) & = & f_1(\mathcal{H}\ x_{11}) \ldots (\mathcal{H}\ x_{1k_1})\ x_{11} \ldots x_{1k_1} \wedge \\
 & \vdots & \hfill \wedge \\
\quad\quad \forall x_{n1} \ldots x_{nk_n}.\ \mathcal{H}(\mathsf{C}_n\ x_{n1} \ldots x_{nk_n}) & = & f_n(\mathcal{H}\ x_{n1}) \ldots (\mathcal{H}\ x_{nk_n})\ x_{n1} \ldots x_{nk_n}.
\end{array}$$

The theorem embodies the primitive recursion principle for functions over the specified type. It is often referred to as an *initiality* theorem, because of its close relationship to the initiality property used in algebraic specification [108]. The usual properties that can be derived from this theorem are (for discussion on the automation of these proofs, see [71]):

Injectivity of constructors. For each constructor C_i the following holds:

$$(\mathsf{C}_i\ x_{i1} \ldots x_{ik_i} = \mathsf{C}_i\ y_{i1} \ldots y_{ik_i}) = (x_{i1} = y_{i1}) \wedge \ldots \wedge (x_{ik_i} = y_{ik_i}).$$

Distinctness of constructors. For $i \neq j$ the following holds:

$$\neg(\mathsf{C}_i\ \overline{x} = \mathsf{C}_j\ \overline{y}).$$

Structural induction. The following induction theorem can be proved, which allows P to be assumed for an immediate subcomponent v of an application of C_i, provided v has type $(\alpha_1, \ldots, \alpha_n)\ ty$.

$$\begin{array}{ll}
\forall P.\ (\forall \overline{x}.\ P\ x_1 \wedge \ldots \wedge P\ x_{k_1} \supset P\ (\mathsf{C}_1 \overline{x})) & \wedge \\
\qquad\qquad \vdots & \wedge \\
\quad\ \ (\forall \overline{x}.\ P\ x_1 \wedge \ldots \wedge P\ x_{kn} \supset P\ (\mathsf{C}_n \overline{x})) & \\
\quad\ \ \supset & \\
\quad\quad\ \ \forall x : ty.\ P\ x &
\end{array}$$

However, there are further facts definable and derivable from initiality that we need in the course of our work. In particular, the following theorems will be required by TFL for a datatype: exhaustion, case definition, and case congruence.

Exhaustion

The following theorem schema says that all elements of the datatype are tagged with a constructor. In our work, it is used to derive custom induction theorems, as well as to prove the case congruence theorem for the datatype.

$$\forall x : ty.\ (\exists \overline{y}.x = \mathsf{C}_1\overline{y}) \vee \ldots \vee (\exists \overline{y}.x = \mathsf{C}_n\overline{y})$$

Proof. By structural induction. After induction, the nth disjunct of the nth subgoal is easy to prove since it is just a, perhaps existentially quantified, instance of reflexivity.
□

Case definition

Case expressions for user-defined datatypes are central to our approach, since recursion equations are translated into a nested case expression before a definition of the function is made. The *case* definition is easy to make, being a non-recursive instance of the primitive recursive definition principle for the type.

$$\begin{array}{lcl}(\forall \overline{x}.\ \mathsf{case\_}ty\ f_1 \ldots f_n\ (\mathsf{C}_1\overline{x}) & \equiv & f_1\overline{x}) \\ \quad \wedge\ \ldots\ \wedge & & \\ (\forall \overline{x}.\ \mathsf{case\_}ty\ f_1 \ldots f_n\ (\mathsf{C}_n\overline{x}) & \equiv & f_n\overline{x})\end{array}$$

Case congruence

TFL uses congruence theorems as a means of tracking the context of recursive calls in function definitions. Therefore, it is important that a congruence theorem be proved for every case statement, *i.e.*, for each datatype definition. For each datatype, an instance of the following schema can be computed and proved.

$$\begin{array}{l}(M = N) \wedge \\ (\forall \overline{x}.\ (N = \mathsf{C}_1\overline{x}) \supset f_1\overline{x} = f_1'\overline{x}) \\ \wedge \ldots \wedge \\ (\forall \overline{x}.\ (N = \mathsf{C}_n\overline{x}) \supset f_n\overline{x} = f_n'\overline{x}) \\ \quad \supset \\ \quad \mathsf{case\_ty}\ f_1 \ldots f_n\ M = \mathsf{case\_ty}\ f_1' \ldots f_n' N.\end{array}$$

Proof. Assume the antecedents of the proposition and perform a case analysis on N, using the exhaustion theorem for the datatype. For each case i, we expand the case definition and apply f_i and f_i' to the arguments of C_i, which, by assumption, are equal.
□

2.2.1 Common datatypes

Since we will make frequent use of the datatypes of booleans, pairs, numbers, lists, and options, we now define the constructors and some common operations for these types. Further definitions can be also be made automatically for each type, *e.g.*, fold and map, but we shall omit these.

Truth values

The datatype bool is built from the constructors True and False.

Initiality	$\forall e_0 e_1 : \alpha.\ \exists! f.\ (f\ \mathsf{True} = e_0) \wedge (f\ \mathsf{False} = e_1)$
Injectivity	(not applicable: nullary constructors)
Distinctness	$\neg(\mathsf{True} = \mathsf{False})$
Nchotomy	$\forall b.\ (b = \mathsf{True}) \vee (b = \mathsf{False})$
Induction	$\forall P.\ P\ \mathsf{True} \wedge P\ \mathsf{False} \supset \forall b.\ P\ b$
Case expression	$\mathsf{bool\_case}\ x\ y\ \mathsf{True} = x$ $\mathsf{bool\_case}\ x\ y\ \mathsf{False} = y$
Distribution	$\forall P.\ P\ (\mathsf{bool\_case}\ x\ y\ b) = \mathsf{bool\_case}\ (P\ x)\ (P\ y)\ b$

The familiar conditional construct can be defined as:

$$\texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ y \equiv \mathsf{bool\_case}\ x\ y\ b.$$

Pairs

The datatype (α, β) pair is built from the comma (,)constructor. A type (τ, δ) pair may also be written $\tau * \delta$.

Initiality	$\forall f.\ \exists! h.\ \forall x\ y.\ h\ (x, y) = f\ x\ y$
Injectivity	$\forall x\ y\ a\ b.\ ((x, y) = (a, b)) = (x = a) \wedge (y = b)$
Distinctness	(not applicable: only one constructor)
Nchotomy	$\forall p.\ \exists x\ y.\ p = (x, y)$
Induction	$\forall P.\ (\forall x\ y.\ P\ (x, y)) \supset \forall p.\ P\ p$
Case expression	$\mathsf{pair\_case}\ f\ (x, y) = f\ x\ y$
Distribution	$\forall g.\ g\ (\mathsf{pair\_case}\ f\ p) = \mathsf{pair\_case}\ (\boldsymbol{\lambda} p_1 p_2.\ g(f\ p_1\ p_2))\ p$

The functions fst and snd return the first and second elements of a pair respectively:

$$\begin{array}{rcl} \mathsf{fst}\ (x,y) & \equiv & x \\ \mathsf{snd}\ (x,y) & \equiv & y \\ (\mathsf{fst}\ x, \mathsf{snd}\ x) & = & x \end{array}$$

The following is a useful infix combinator:

$$(f\#g)(x,y) \equiv (f\ x, g\ y).$$

The type of this combinator is $(\alpha \rightarrow \beta) \rightarrow (\delta \rightarrow \gamma) \rightarrow \alpha * \delta \rightarrow \beta * \gamma$.

Pairing can be introduced into the syntax of lambda binding by use of pair_case. The concrete syntax $\boldsymbol{\lambda}(x,y).\ M$, meant to denote a $\boldsymbol{\lambda}$-abstraction over pairs, can be represented by $\mathsf{pair\_case}(\boldsymbol{\lambda}x\ y.\ M)$. The notion of β-redex can be similarly lifted:

$$\mathsf{pair\_case}(\boldsymbol{\lambda}x\ y.\ M)\ (p,q)$$

with paired-β reduction being implemented by applying the definition of pair_case, followed by β-reduction:

$$\begin{array}{rcl} (\mathsf{pair\_case}(\boldsymbol{\lambda}x\ y.\ M))\ (p,q) & = & (\boldsymbol{\lambda}x\ y.\ M)\ p\ q \\ & =_\beta & [y \mapsto q]([x \mapsto p]M). \end{array}$$

We will later see how this treatment can be generalized to arbitrary datatypes via a translation from patterns to nested case expressions.

Paired lets are also important, since they allow the modelling of multiple simultaneous binding: for example

$$\mathsf{let}\ (x,y) = M\ \mathsf{in}\ N$$

is represented by $\mathsf{Let}(\mathsf{pair\_case}(\boldsymbol{\lambda}x\ y.\ N))\ M$.

Sums

The datatype (α,β) sum is built from the INL and INR constructors. A type (τ,δ) sum may also be written $\tau + \delta$.

Initiality	$\forall f\ g.\ \exists! h.\ (\forall x.\ h\ (\mathsf{INL}\ x) = f\ x) \wedge (\forall y.\ h\ (\mathsf{INR}\ y) = g\ y)$
Injectivity	$(\forall x_1 x_2.\ (\mathsf{INL}\ x_1 = \mathsf{INL}\ x_2) = (x_1 = x_2)) \wedge$ $(\forall y_1 y_2.\ (\mathsf{INR}\ y_1 = \mathsf{INR}\ y_2) = (y_1 = y_2))$
Distinctness	$\forall x\ y.\ \neg(\mathsf{INL}\ x = \mathsf{INR}\ y)$
Nchotomy	$\forall s.\ (\exists x.\ s = \mathsf{INL}\ x) \vee (\exists y.\ s = \mathsf{INR}\ y)$
Induction	$\forall P.\ (\forall x.\ P\ (\mathsf{INL}\ x)) \wedge (\forall y.\ P\ (\mathsf{INR}\ x)) \supset \forall s.\ P\ s$
Case expression	$\mathsf{sum\_case}\ f\ g\ (\mathsf{INL}\ x) = f\ x$ $\mathsf{sum\_case}\ f\ g\ (\mathsf{INR}\ y) = g\ y$
Distribution	$\forall h.\ h(\mathsf{sum\_case}\ f\ g\ s) = \mathsf{sum\_case}\ (\boldsymbol{\lambda} x.h(f\ x))\ (\boldsymbol{\lambda} y.h(g\ y))\ s$

INL and INR have partial inverses OUTL and OUTR:

$$\mathsf{OUTL}(\mathsf{INL}\ x) = x,$$
$$\mathsf{OUTR}(\mathsf{INR}\ x) = x.$$

Natural numbers

The datatype nat is built from the constructors 0 and Suc.

Initiality	$\forall e\ f.\ \exists! g.\ (g\ 0 = e) \wedge (\forall n.\ g\ (\mathsf{Suc}\ n) = f\ (g\ n)\ n)$
Injectivity	$\forall m\ n.\ (\mathsf{Suc}\ m = \mathsf{Suc}\ n) = (m = n)$
Distinctness	$\forall n.\ \neg(\mathsf{Suc}\ n = 0)$
Nchotomy	$\forall m.\ (m = 0) \vee (\exists n.\ m = \mathsf{Suc}\ n)$
Induction	$\forall P.\ P\ 0 \wedge (\forall n.\ P\ n \supset P\ (\mathsf{Suc}\ n)) \supset \forall n.\ P\ n$
Case expression	$\mathsf{num\_case}\ b\ f\ 0 = b$ $\mathsf{num\_case}\ b\ f\ (\mathsf{Suc}\ n) = f\ n$
Distribution	$\forall g.\ g(\mathsf{num\_case}\ b\ f\ n) = \mathsf{num\_case}\ (g\ b)\ (\boldsymbol{\lambda} m.\ g(f\ m))\ n$

Another useful induction principle for numbers—interderivable with mathematical induction—is *strong*, also known as *complete*, also known as *course-of-values* induction:

$$\forall P.\ (\forall m.\ (\forall k.\ k < m \supset P\ k) \supset P\ m) \supset \forall n.\ P\ n.$$

Lists

The datatype α list is built from :: (cons) and [] (nil).

Initiality	$\forall x\ f.\exists! g.\ (g\ [\,] = x) \land (\forall h\ t.\ g\ (h :: t) = f\ (g\ t)\ h\ t)$
Injectivity	$\forall h\ t\ h'\ t'.\ (h :: t = h' :: t') = (h = h') \land (t = t')$
Distinctness	$\forall h\ t.\ \neg([\,] = h :: t)$
Nchotomy	$\forall l.\ (l = [\,]) \lor (\exists h\ t.\ l = h :: t)$
Induction	$\forall P.\ P\ [\,] \land (\forall h\ t.\ P\ t \supset P\ (h :: t)) \supset \forall l.\ P\ l$
Case expression	$\mathsf{list\_case}\ b\ f\ [\,] = b$ $\mathsf{list\_case}\ b\ f\ (h :: t) = f\ h\ t$
Distribution	$\forall g.\ g\ (\mathsf{list\_case}\ b\ f\ l) = \mathsf{list\_case}\ (g\ b)(\boldsymbol{\lambda} h\ t.\ g\ (f\ h\ t))\ l$

The familiar list-processing functions mem, filter, length, @, map, exists, and rev_itlist (also known as 'foldl') are used in some examples; they are defined as follows:

$$\begin{aligned}
\mathsf{mem}\ x\ [\,] &\equiv \mathsf{False}\\
\mathsf{mem}\ x\ (h :: t) &\equiv (x = h) \lor \mathsf{mem}\ x\ t\\[1em]
\mathsf{filter}\ P\ [\,] &\equiv [\,]\\
\mathsf{filter}\ P\ (h :: t) &\equiv \mathtt{if}\ P\ h\ \mathtt{then}\ h :: (\mathsf{filter}\ P\ t)\ \mathtt{else}\ \mathsf{filter}\ P\ t\\[1em]
\mathsf{length}\ [\,] &\equiv \mathsf{0}\\
\mathsf{length}\ (h :: t) &\equiv \mathsf{1} + \mathsf{length}\ t\\[1em]
[\,]\ @\ l &\equiv l\\
(h :: t)\ @\ l &\equiv h :: (t\ @\ l)\\[1em]
\mathsf{map}\ f\ [\,] &\equiv [\,]\\
\mathsf{map}\ f\ (h :: t) &\equiv f\ h :: \mathsf{map}\ f\ t\\[1em]
\mathsf{exists}\ P\ [\,] &\equiv \mathsf{False}\\
\mathsf{exists}\ P\ (h :: t) &\equiv P\ h \lor \mathsf{exists}\ P\ t\\[1em]
\mathsf{rev\_itlist}\ f\ [\,]\ v &\equiv v\\
\mathsf{rev\_itlist}\ f\ (h :: t)\ v &\equiv \mathsf{rev\_itlist}\ f\ t\ (f\ h\ v)
\end{aligned}$$

Option

The datatype α option is built from None and Some.

Initiality	$\forall e\ f.\ \exists! g.\ (g\ \mathsf{None} = e) \land (\forall x.\ g\ (\mathsf{Some}\ x) = f\ x)$
Injectivity	$\forall x\ x'.\ (\mathsf{Some}\ x = \mathsf{Some}\ x') = (x = x')$
Distinctness	$\forall x.\ \neg(\mathsf{None} = \mathsf{Some}\ x)$
Nchotomy	$\forall opt.\ (opt = \mathsf{None}) \lor \exists x.\ opt = \mathsf{Some}\ x$
Induction	$\forall P.\ P\ \mathsf{None} \land (\forall x.\ P\ (\mathsf{Some}\ x)) \supset \forall opt.\ P\ opt$
Case expression	$\mathsf{option\_case}\ u\ f\ \mathsf{None} = u$ $\mathsf{option\_case}\ u\ f\ (\mathsf{Some}\ x) = f\ x$
Distribution	$\forall g.\ g\ (\mathsf{option\_case}\ b\ f\ o) = \mathsf{option\_case}\ (g\ b)\ (\boldsymbol{\lambda} x. g\ (f\ x))\ o$

2.3 Wellfoundedness and induction

Our approach to defining recursive functions bases itself on the idea of *wellfoundedness*, which is a notion from set theory. The decreasing paths in a wellfounded relation are all of finite length. In the context of programs, this means that, if the arguments to recursive calls in a function definition can be placed in a wellfounded relation, then the function will terminate. In the context of logic, the corresponding notion is that the function is total. There are several equivalent formal definitions of wellfoundedness [92]; the following is quite easy to deal with (it asserts that R is wellfounded iff every non-empty set has an R-minimal element):[3]

Definition 5 (Wellfoundedness)

$$WF(R) \equiv \forall P.\ (\exists w.\ P\ w) \supset \exists min.\ P\ min \land \forall b.\ R\ b\ min \supset \neg P\ b.$$

We will sometimes use the phrases *R-less* or *R-smaller* as an infix way of speaking about a wellfounded relation R, *e.g.*, x is R-less than y.

From the definition, one can quickly prove a general induction theorem.

Theorem 6 (Wellfounded induction)

$$\forall P\ R.\ \mathsf{WF}(R) \supset (\forall x.\ (\forall y.\ R\ y\ x \supset P\ y) \supset P\ x) \supset \forall x.\ P\ x.$$

[3] Note that there need not be a single R-minimal element in a set.

Proof. Assume R is wellfounded. Assume $P\ x$ holds when $P\ y$ holds for all y R-less than x. Towards a contradiction, assume there's a z such that $\neg P\ z$. By wellfoundedness, there's a R-minimal element min such that $\neg P\ min$. Hence, for all y R-less than min, $P\ y$ holds. Hence $P\ min$ holds, a contradiction.
□

One can think of the idea of wellfoundedness as being a means of identifying when induction is valid: wellfoundedness ensures that the inductive hypothesis is not, in effect, what is to be proved. Wellfounded induction generalizes the usual mathematical induction in two ways: it can be instantiated to any predicate, not just predicates on numbers; and it allows the assumption of strong induction hypotheses, *i.e.*, one is allowed to assume that the property holds for *all* R-smaller elements, not just the predecessor. Many common induction theorems *e.g.*, mathematical, course-of-values, structural, simultaneous, mutual, transfinite, and recursion induction, can be obtained as instantiations of this one general result.

Historically, mathematical induction originated with Dedekind [30] (who called it complete induction, somewhat confusingly for modern readers) and also Peano. Induction along the membership relation ($\in$-induction) was established early in the development of set theory. The generalization to induction over arbitrary wellfounded relations came in Shepherdson's work on inner models of set theory in the early 1950's.[4]

2.4 Transitive closure

The proof of the recursion theorem requires some facts about transitive closures.

Definition 7 (Transitive)

$$\mathsf{transitive}\ R \equiv \forall x\ y\ z.\ R\ x\ y \land R\ y\ z \supset R\ x\ z.$$

The transitive closure of a relation $R : \alpha \to \alpha \to \mathsf{bool}$ can be inductively defined:

Definition 8 (Transitive closure)

$$\begin{aligned}\mathsf{TC}\ R\ a\ b \equiv \forall P.\ & (\forall x\ y.\ R\ x\ y \supset P\ x\ y) \land \\ & (\forall x\ y\ z.\ P\ x\ y \land P\ y\ z \supset P\ x\ z) \\ & \supset P\ a\ b.\end{aligned}$$

The elements of a transitive closure obey the rules in the inductive definition:

Theorem 9 $\mathsf{transitive}(\mathsf{TC}\ R) \land (R\ x\ y \supset \mathsf{TC}\ R\ x\ y)$.

One can prove properties of a transitive closure by induction on its construction:

[4] Thanks to Ake Kanamori and Thomas Forster for this tidbit.

Theorem 10 (Transitive closure induction)

$$\begin{array}{ll} \forall R\ P. & (\forall x\ y.\ R\ x\ y \supset P\ x\ y) \land \\ & (\forall x\ y\ z.\ P\ x\ y \land P\ y\ z \supset P\ x\ z) \\ & \supset \forall u\ v.\ (\mathsf{TC}\ R)\ u\ v \supset P\ u\ v. \end{array}$$

Proof. Direct from the definition of transitive closure.
□
In a certain sense, it is possible to drop down out of a transitive closure:

Lemma 11 $\mathsf{TC}\ R\ x\ z \supset R\ x\ z \lor \exists y.\ \mathsf{TC}\ R\ x\ y \land R\ y\ z.$

Proof. By transitive closure induction.
□
Transitive closure propagates wellfoundedness.

Theorem 12 $\mathsf{WF}(R) \supset \mathsf{WF}(\mathsf{TC}\ R).$

Proof. By contraposition, assume that $\mathsf{TC}\ R$ is not wellfounded. Let B be the non-empty set with no $\mathsf{TC}\ R$-minimal element. We must show the existence of a non-empty set with no R-minimal element. One such set is described by the formula

$$\lambda m. \exists a\ z.\ B\ a \land \mathsf{TC}\ R\ a\ m \land \mathsf{TC}\ R\ m\ z \land B\ z. \tag{2.1}$$

Let w be an element of B. By two applications of the non-wellfoundedness of $\mathsf{TC}\ R$ we obtain two elements of B, w_1 and w_2, such that $\mathsf{TC}\ R\ w_1\ w$ and $\mathsf{TC}\ R\ w_2\ w_1$. Thus (2.1) is non-empty; we now show that it has no R-minimal element. Let x be an element of (2.1); therefore, there exists a and z in B such that $\mathsf{TC}\ R\ a\ x$ and $\mathsf{TC}\ R\ x\ z$. Applying Lemma 11 to the former, there are two cases to consider:

$\underline{R\ a\ x}$**.** We have that a is R-smaller than x. It remains to show that a is in (2.1). Since a is in B, there is an element b in B such that $\mathsf{TC}\ R\ b\ a$. It remains to show $\mathsf{TC}\ R\ a\ z$; this is immediate by the transitivity of $\mathsf{TC}\ R$.

$\underline{\exists y.\ \mathsf{TC}\ R\ a\ y \land R\ y\ x}$**.** We have that y is R-smaller than x. Taking a and z to be our endpoints, it remains to show $\mathsf{TC}\ R\ y\ z$; and this is easy, by Lemma 11.

□

2.5 Wellfounded Recursion

Wellfoundedness not only justifies a general induction theorem, it also justifies a general recursion theorem. Recursion theorems typically assert the unique existence of functions described by recursive equations. Their importance is that they provide a sound way of interpreting a self-referential finite description (program) as an infinite object (function) in the logic. In this section we derive a version of the wellfounded recursion theorem that Tobias Nipkow initially proved in the Isabelle system. Schwichtenberg and Wainer seem to have independently arrived at a similar formulation [93]. We will give a leisurely discussion of the theorem, with the intent being to give a proof so detailed that it can be re-created even by those who do not understand it!

The statement of the wellfounded recursion theorem uses a ternary operator that restricts a function to a certain set of values.

Definition 13 (Restriction)

$$(f \,|\, R, y) \equiv \lambda x.\ \mathtt{if}\ R\ x\ y\ \mathtt{then}\ f\ x\ \mathtt{else}\ \mathsf{Arb}.$$

Theorem 14 $R\ x\ y \supset (f \,|\, R, y)\ x = f\ x.$

Theorem 15 $\neg R\ x\ y \supset (f \,|\, R, y)\ x = \mathsf{Arb}.$

In set theory, or other logics of partial functions, function restriction may result in a partial function. In a logic of total functions, such as HOL, a restriction of a function is still a total function, giving a fixed but arbitrary value when applied outside of the restriction. This is an instance of the use of *underspecification* to avoid the use of partial functions.

We continue the development by defining the set of *attempts* for a functional M at argument x in the presence of relation R:

Definition 16 (Attempts)

$$\mathsf{attempt}\ R\ M\ x\ f \equiv (\forall w.\ f\ w = \mathtt{if}\ R\ w\ x\ \mathtt{then}\ M\ (f \,|\, R, w)\ w\ \mathtt{else}\ \mathsf{Arb}).$$

This definition is a predicate on functions that checks whether they agree with the body of the recursive function at each R-predecessor of x. All other arguments must be mapped to Arb.

By wellfounded induction, any two attempts agree on their common domain:

Theorem 17

$$\begin{array}{l} \mathsf{WF}(R)\ \wedge \\ \mathsf{transitive}(R)\ \wedge \\ \mathsf{attempt}\ R\ M\ u\ f\ \wedge \\ \mathsf{attempt}\ R\ M\ v\ g \\ \quad \supset \forall x. R\ x\ u \wedge R\ x\ v \supset f(x) = g(x). \end{array}$$

The following definition chooses an attempt:

Definition 18 (the_fun)

$$\mathsf{the\_fun}\ R\ M\ x \equiv \varepsilon f.\ \mathsf{attempt}\ R\ M\ x\ f.$$

Given R, M, and x, $\mathsf{the\_fun}\ R\ M\ x$ picks out a function that obeys the recursion at all R-predecessors of x, and delivers Arb otherwise. We can immediately obtain, by application of the Select Axiom:

Lemma 19 $(\exists g.\ \mathsf{attempt}\ R\ M\ x\ g) \supset \mathsf{attempt}\ R\ M\ x\ (\mathsf{the\_fun}\ R\ M\ x)$.

We now prove the crucial lemma: attempts exist.

Theorem 20 (Existence)

$$\mathsf{WF}(R) \land \mathsf{transitive}(R) \supset \forall x.\ \exists f.\ \mathsf{attempt}\ R\ M\ x\ f.$$

Proof. Assume that R is wellfounded and transitive. By wellfounded induction, assume that there exists an attempt for every R-predecessor of x. By lemma 19, this is equivalent to $\forall y.\ R\ y\ x \supset \mathsf{attempt}\ R\ M\ y\ (\mathsf{the\_fun}\ R\ M\ y)$. We claim that

$$\boldsymbol{\lambda} p.\ \mathtt{if}\ R\,p\,x\ \mathtt{then}\ M\ (\mathsf{the\_fun}\ R\ M\ p)\ p\ \mathtt{else}\ \mathsf{Arb}$$

is an x-attempt. Consider an arbitrary w: if it is not a parent of x, then our witness gives Arb, as an x-attempt should. Alternatively, suppose $R\ w\ x$. It remains to show

$$\begin{array}{l} M\ (\mathsf{the\_fun}\ R\ M\ w)\ w = \\ M\ ((\boldsymbol{\lambda} p.\ \mathtt{if}\ R\,p\,x\ \mathtt{then}\ M\ (\mathsf{the\_fun}\ R\ M\ p)\ p\ \mathtt{else}\ \mathsf{Arb}) \,|\, R, w)\ w \end{array}$$

which is equivalent to

$$\mathsf{the\_fun}\ R\ M\ w = (\boldsymbol{\lambda} p.\ \mathtt{if}\ R\,p\,x\ \mathtt{then}\ M\ (\mathsf{the\_fun}\ R\ M\ p)\ p\ \mathtt{else}\ \mathsf{Arb}) \,|\, R, w.$$

By the induction hypothesis, there is a w-attempt $\mathsf{the\_fun}\ R\ M\ w$ and by the definition of attempt, it now suffices to show

$$\begin{array}{l} (\boldsymbol{\lambda} y.\ M\ (\mathsf{the\_fun}\ R\ M\ w \,|\, R, y)\ y) \,|\, R, w = \\ (\boldsymbol{\lambda} p.\ \mathtt{if}\ R\,p\,x\ \mathtt{then}\ M\ (\mathsf{the\_fun}\ R\ M\ p)\ p\ \mathtt{else}\ \mathsf{Arb}) \,|\, R, w. \end{array}$$

This is the equality of two equally-restricted functions: it suffices to consider their equality on v a parent of w. By transitivity, $R\ v\ x$, and if we can show

$$\mathsf{the\_fun}\ R\ M\ w \,|\, R, v = \mathsf{the\_fun}\ R\ M\ v$$

we will be finished. By the induction hypothesis $\mathsf{the\_fun}\ R\ M\ v$ is an attempt and theorem 17 applies.

□

As a corollary, we have the useful theorem for $\mathsf{the\_fun}$:

Lemma 21

$$\begin{array}{l}\mathsf{WF}(R) \wedge \mathsf{transitive}(R) \wedge R\ w\ x \\ \quad\supset \\ \mathsf{the\_fun}\ R\ M\ x\ w = M\ (\mathsf{the\_fun}\ R\ M\ x \,|\, R, w)\ w.\end{array}$$

Our stated desire was to prove the wellfounded recursion theorem and we have now built sufficient basis to do that. There is now a question as to *what* recursion theorem to prove. The standard formulation is the following:

$$\mathsf{WF}(R) \supset \forall M.\ \exists! f.\ \forall x.\ f(x) = M(f \,|\, R, x)\ x.$$

This theorem will be a corollary to our development which states the theorem in terms of a 'controlled' fixpoint combinator.

Definition 22 ($\mathsf{WFREC} : (\alpha \to \alpha \to \mathsf{bool}) \to ((\alpha \to \beta) \to (\alpha \to \beta)) \to \alpha \to \beta$)

$$\mathsf{WFREC}\ R\ M \equiv \boldsymbol{\lambda} x.\ M\ \underbrace{(\mathsf{the\_fun}(\mathsf{TC}\ R)\ (\boldsymbol{\lambda} f\ v.\ M\ (f \,|\, R, v)\ v)\ x \,|\, R, x)}_{*}\ x.$$

We intend $\mathsf{WFREC}\ R\ M$ to denote the recursive function described by M and controlled by R. Given x, it applies M to x and a complicated expression (*) denoting the function applied in recursive calls. The essential idea in the complicated expression is that

$$\mathsf{the\_fun}\ (\mathsf{TC}\ R)\ (\boldsymbol{\lambda} f\ v.\ M\ (f \,|\, R, v)\ v)\ x$$

is a function that obeys the recursion for each R-ancestor of x (given by $\mathsf{TC}(R)$). The expression doing the unfolding, $(\boldsymbol{\lambda} f\ v.\ M\ (f \,|\, R, v)\ v)$, makes sure that each recursive call is on an R-predecessor of the current call. We now check these intuitions.

Theorem 23 (Wellfounded recursion)

$$\mathsf{WF}(R) \supset \forall x.\ \mathsf{WFREC}\ R\ M\ x = M\ (\mathsf{WFREC}\ R\ M \,|\, R, x)\ x.$$

Proof. After expanding the definition of WFREC and some simple function-level reasoning, we must prove the equality of the unfoldings at y, a parent of x:

$$\begin{array}{l}\mathsf{the\_fun}\ (\mathsf{TC}\ R)\ (\boldsymbol{\lambda} f\ v.\ M\ (f \,|\, R, v)\ v)\ x\ y\ = \\ M\ (\mathsf{the\_fun}\ (\mathsf{TC}\ R)\ (\boldsymbol{\lambda} f\ v.\ M\ (f \,|\, R, v)\ v)\ y \,|\, R, y)\ y.\end{array}$$

By use of Lemma (21) and the lemma $\vdash ((f \,|\, \mathsf{TC}\ R, w) \,|\, R, w) = (f \,|\, R, w)$, this reduces to showing

$$\begin{array}{l}\mathsf{the\_fun}\ (\mathsf{TC}\ R)\ (\boldsymbol{\lambda} f\ v.\ M\ (f \,|\, R, v)\ v)\ x\ w\ = \\ \mathsf{the\_fun}\ (\mathsf{TC}\ R)\ (\boldsymbol{\lambda} f\ v.\ M\ (f \,|\, R, v)\ v)\ y\ w\end{array}$$

for w a parent of y (and hence an ancestor of both x and y). The functions being applied on both sides are both attempts and w is in their common domain so Theorem 17 applies.
□

The theorem that TFL manipulates is just a common subexpression contraction of Theorem 23.

Theorem 24 (TFL recursion)

$$(f = \mathsf{WFREC}\ R\ M) \supset \mathsf{WF}(R) \supset \forall x.\ f(x) = M\ (f \,|\, R, x)\ x.$$

□

The standard formulation of the wellfounded recursion theorem is proved using Theorem 23 for existence, and proving uniqueness by wellfounded induction.

Theorem 25

$$\mathsf{WF}(R) \supset \forall M.\ \exists! f.\ \forall x.\ f(x) = M(f \,|\, R, x)\ x.$$

□

2.6 A collection of wellfounded relations

Defining recursive functions in HOL always means proving termination (totality). When using Theorem 24, a wellfounded relation must be supplied that will justify the totality of the function: if all the recursive calls can be shown to be in the relation, then the unconstrained recursion equations, as given by the user, can be validly used in further proof. It can be quite difficult to directly establish that a particular relation is wellfounded. However, this situation rarely occurs since it is possible to combine basic wellfounded relations into more powerful wellfounded relations. Our current collection of these *wellfoundedness operators* comprises lexicographic combination, inverse image, subsets, transitive closure, and the multiset extension. These operators give a powerful and extensible language for expressing termination arguments.

Note that this section does not form a *requirement* for the machinery of TFL. Instead, it should be thought of as the humble core of a library of wellfounded relations that may be useful for proving termination of functions defined by the machinery developed in later chapters.

Theorem 26 (Empty) $\mathsf{WF}(\lambda x\ y.\ \mathsf{False})$.

Proof. Immediate, by the definition of wellfoundedness.
□

The empty relation is wellfounded. This apparently useless fact can be employed to extend a definition tool for recursive functions to definitions of non-recursive functions: when a non-recursive definition is made, the empty relation can be given as a termination relation.

Theorem 27 (Subrelation) $\mathsf{WF}(R) \land (\forall x\, y.\ Q\ x\ y \supset R\ x\ y) \supset \mathsf{WF}(Q)$.

Proof. By the wellfoundedness of R, an arbitrary non-empty set P has an R-minimal element *min*. This element is also Q-minimal, for suppose there was an element b of P that is Q-smaller than *min*. Then $R\ b\ min$ and b is not an element of P. Contradiction.
□

Definition 28 (Lexicographic product)

$$\mathsf{LEX}\ R_1\ R_2\ (u, v)\ (x, w) \equiv R_1\ u\ x \lor (u = x \land R_2\ v\ w).$$

Theorem 29 $\mathsf{WF}(R) \land \mathsf{WF}(Q) \supset \mathsf{WF}\ (\mathsf{LEX}\ R\ Q)$.

Proof. A minimal element of the product is found by first finding the set of first elements of the pairs in the product. That set is non-empty, so it has an R-minimal element; call it min_0. Find the set of second elements of those pairs in the product whose first element is min_0. This set is non-empty, so it has a Q-minimal element; call it min_1. A minimal element of the product is (min_0, min_1).
□

Definition 30 (Relational product)

$$\mathsf{RPROD}\ R_1\ R_2(u, v)\ (x, w) \equiv R_1\ u\ x \land R_2\ v\ w.$$

The wellfoundedness of the relational product is an application of subrelation to the lexicographic product.

Theorem 31 $\mathsf{WF}(R) \land \mathsf{WF}(Q) \supset \mathsf{WF}(\mathsf{RPROD}\ R\ Q)$.

Proof. By Theorems 27 and 29.
□
The following construction is probably the most important means of building compound wellfounded relations.

Definition 32 (Inverse image)

$$\mathsf{inv\_image}\ (R : \beta \to \beta \to \mathsf{bool})\ (f : \alpha \to \beta) \equiv \boldsymbol{\lambda} x\ y.\ R\ (f\ x)\ (f\ y)$$

Theorem 33 $\mathsf{WF}(R) \supset \mathsf{WF}(\mathsf{inv\_image}\ R\ f)$.

Proof. For notational convenience, let R_1 stand for $\mathsf{inv\_image}\ R\ f$. Consider a non-empty subset $\mathcal{A}$ of α (this always exists because type variables denote non-empty sets in HOL). The image of f over $\mathcal{A}$ (call it $\mathcal{B}$) is non-empty, because f is total over α. By wellfoundedness, $\mathcal{B}$ has an R-minimal element *min*. By construction, there is an x in $\mathcal{A}$ such that $f\ x = min$. We claim that x is an

R_1-minimal element of $\mathcal{A}$, and show this by contradiction. Suppose there is a y in $\mathcal{A}$ such that $R_1\ y\ x$. Then $R(f\ y)(f\ x)$, *i.e.*, $R(f\ y)\ min$. By the R-minimality of min, $f\ y$ is not in $\mathcal{B}$. If $f\ y$ is not in $\mathcal{B}$, then y is not in $\mathcal{A}$, since $\mathcal{B}$ is just the image of f over $\mathcal{A}$.
□

In the following, we require a theory of multisets (sets with repeated elements) having operations for membership ($\in_{\mathcal{M}}$), union ($\cup_{\mathcal{M}}$), difference ($-_{\mathcal{M}}$), and a finiteness predicate (Fin). The empty multiset is written $\{\,\}_{\mathcal{M}}$ and a multiset containing the single element x is written $\{x\}_{\mathcal{M}}$. In unambiguous situations, the $\mathcal{M}$ subscript will be dropped. An R-predecessor M to a multiset N is obtained by taking an element x out of N and replacing it with a finite multiset Y of R-smaller objects.

Definition 34 (Multiset predecessor)

$$\begin{array}{ll}\mathsf{predMset}\ R\ M\ N \equiv \exists x\ Y. & x \in_{\mathcal{M}} N\ \wedge \\ & (M = (N - \{x\}_{\mathcal{M}}) \cup Y)\ \wedge \\ & (\forall y.\ y \in_{\mathcal{M}} Y \supset R\ y\ x)\ \wedge \\ & \mathsf{Fin}\ N \wedge \mathsf{Fin}\ Y\end{array}$$

Theorem 35 $\mathsf{WF}(R) \supset \mathsf{WF}\ (\mathsf{predMset}\ R)$.

Proof. There are two proofs for this. The original, due to Dershowitz and Manna, in outline, is the following: towards a contradiction, suppose $\mathsf{predMset}\ R$ is not wellfounded, and that R is wellfounded. There is an infinite descending $\mathsf{predMset}\ R$ sequence. From this sequence, a labelled tree is built such that each child of a node has a label that is R-smaller than the label of the node. By construction, the tree is infinite and finitely branching, thus has an infinite path by Koenig's lemma. Contradiction. The author has formalized the Dershowitz and Manna proof, and found it to be a challenging task.

There is a recent and logically simpler proof (it doesn't require Koenig's lemma) credited to Bucholz and independently Coquand, which makes clever use of inductive definitions. One formalization of this recent proof can be found in [88, paper II]: it is relatively short compared to the original but much less intuitive, in our opinion.
□

A helpful definition is the following map from lists to multisets.

Definition 36 (list_to_mset)

$$\begin{array}{rcl}\mathsf{list\_to\_mset}[\,] & \equiv & \{\,\}_{\mathcal{M}} \\ \mathsf{list\_to\_mset}(h :: t) & \equiv & \{h\}_{\mathcal{M}} \cup (\mathsf{list\_to\_mset}\ t)\end{array}$$

Thus far, the wellfounded relations we have discussed have all been abstract. Turning to more concrete relations, the predecessor relation on numbers is wellfounded.

Definition 37 (Predecessor) $\mathsf{pred}\ x\ y \equiv (y = \mathsf{Suc}\ x)$.

Theorem 38 $\mathsf{WF\ pred}$.

Proof. Suppose not; then there's a non-empty set of numbers P with no pred-minimal element. However, an induction shows that P does have a pred-minimal element. Contradiction.
□

The following instantiation of the inverse image combinator is heavily used in automating proofs of termination of programs.

Definition 39 (Measure) $\mathsf{measure} \equiv \mathsf{inv\_image}\ (<)$.

Theorem 40 $\forall f.\ \mathsf{WF}\ (\mathsf{measure}\ f)$.

Proof. If $<$ is defined as $\mathsf{TC}\ (\boldsymbol{\lambda} x\ y. y = \mathsf{Suc}\ x)$, the proof is immediate, by Theorems 12 and 38.
□

For the following, we require a theory of sets defining finiteness of sets and the notion of proper subset, as well as a theorem relating proper subsets and cardinality.

Definition 41 (insert)

$$\mathsf{insert}\ x\ S \equiv \{y \mid (y = x) \lor y \in S\}.$$

Definition 42 (finite set)

$$\mathsf{finite}\ S \equiv \forall P.\ P\{\ \} \land (\forall x\ s.\ P(s) \supset P(\mathsf{insert}\ x\ s)) \supset P\ S.$$

Definition 43 (Proper (finite) subset)

$$\mathsf{fpss}\ S_1\ S_2 \equiv \mathsf{finite}\ S_2 \land (S_1 \neq S_2) \land \forall x.\ S_1\ x \supset S_2\ x.$$

Theorem 44 $\mathsf{fpss}\ S_1\ S_2 \supset \mathsf{card}(S_1) < \mathsf{card}(S_2)$.

Proof. By induction on the construction of S_2.
□

Theorem 45 $\mathsf{WF(fpss)}$.

Proof. By use of Theorems 27, 40, and 44.
□

2.6.1 Wellfounded relations for datatypes

When a datatype is defined, new wellfounded relations can be defined for it. One particularly simple thing to do is to define the 'predecessor' relation for the type, in which every application of a recursive constructor is strictly greater than each of its immediate subterms (of the same type). The proof of wellfoundedness for the predecessor relation for a datatype is easy to automate, but it turns out that the *size* measure for a datatype is more useful, and trivial to prove wellfounded, by Theorem 40. Size measures are extremely useful for automation of termination proofs.

Thinking of an element of a datatype as a tree, it is standard to define its size as one plus the sum of the sizes of the subtrees (leaves are assigned a size of zero). Notice, however, that in a simply-typed setting, each datatype needs its own size function. Another problem is how to define the size of polymorphic datatypes, since the size of an instance of the type is not well captured by the size of the defined type. For example, the size of a polymorphic type of lists could be taken to be the length of the list, but what is the size of a list of numbers, or a list of lists?

Problems of this kind can sometimes be solved by 'interpreting' each type variable by a function. In particular, the notion of size can be parameterized so that the size of a term of polymorphic type can take account of the size of possible instantiations of the type variables. Thus, we intend to define the size of elements of type $(\alpha_1, \ldots, \alpha_n)\tau$ as a higher order function parameterized by n size functions:

$$\tau\_\mathsf{size}\ (f_1 : \alpha_1 \to \mathsf{num}) \ldots (f_n : \alpha_n \to \mathsf{num})\ (x : (\alpha_1, \ldots, \alpha_n)\tau) \equiv \ldots$$

Such a definition depends on a metalanguage translation $(|_|_{\Gamma,\Theta})$that maps a HOL type τ to a HOL term having type $\tau \to \mathsf{num}$. The general idea is to traverse the type, and build a term by replacing type operators by size functions (by use of Γ), and type variables by parameters (by use of Θ). Thus Γ maps previously defined type operators to their associated size functions, and Θ maps $\alpha_1, \ldots, \alpha_n$ to $f_1, \ldots, f_n$.

Definition 46 (Type size)

$$\begin{array}{rcl} |v|_{\Gamma,\Theta} & \equiv & \Theta(v) \quad \textit{when } v \textit{ is a type variable} \\ |(\tau_1, \ldots, \tau_j)\ \tau|_{\Gamma,\Theta} & \equiv & \Gamma(\tau)\ (|\tau_1|_{\Gamma,\Theta}) \ldots (|\tau_j|_{\Gamma,\Theta}). \end{array}$$

□

A typical Γ would include at least the following:

type	size expression
$\tau_1 \rightarrow \tau_2$	$\boldsymbol{\lambda} v : \tau_1 \rightarrow \tau_2.\ 0$
$\tau_1 * \tau_2$	$\boldsymbol{\lambda}(f : \tau_1 \rightarrow \mathsf{num})\ (g : \tau_2 \rightarrow \mathsf{num})\ (x, y).\ f\ x + g\ y$
$\tau_1 + \tau_2$	$\mathsf{sum\_case} : (\tau_1 \rightarrow \mathsf{num}) \rightarrow (\tau_2 \rightarrow \mathsf{num}) \rightarrow \tau_1 + \tau_2 \rightarrow \mathsf{num}$
bool	$\mathsf{bool\_case}\ 0\ 0 \quad (\equiv \boldsymbol{\lambda} b.\ 0)$
num	$\boldsymbol{\lambda} x : \mathsf{num}.\ x$
option	$\boldsymbol{\lambda} f.\ \mathsf{option\_case}\ 0\ (\boldsymbol{\lambda} x.\ \mathsf{Suc}\ (f\ x))$

Remarks. The definition of the size of a pair is just the sum of the sizes of the two projections: since pairs are not recursive, it is not useful (for our purposes) to add one to the sum. The size of an element of a sum type is just the size of the injected item: since sum constructors are used as discriminatory tags, any nesting of INL and INR should be ignored in the computation of an object's size.

Examples. In the following, $\Theta \equiv \{\alpha \mapsto f_1, \beta \mapsto f_2\}$:

$$\begin{aligned}
|\mathsf{num} * \mathsf{bool}|_{\Gamma,\Theta} &= (\boldsymbol{\lambda} f\ g\ (x, y).\ f\ x + g\ y)\ (\boldsymbol{\lambda} x.x)(\mathsf{bool\_case}\ 0\ 0) \\
&=_\beta \boldsymbol{\lambda}(x, y).\ x + \mathsf{bool\_case}\ 0\ 0\ y) \\
&= \boldsymbol{\lambda}(x, y).\ x
\end{aligned}$$

$$\begin{aligned}
&|(\alpha * \beta\ \mathsf{option}) + \mathsf{num}|_{\Gamma,\Theta} \\
&\quad = \ \mathsf{sum\_case}\ ((\boldsymbol{\lambda} f\ g\ (x, y).\ f\ x + g\ y)\ f_1 \\
&\qquad\qquad ((\boldsymbol{\lambda} f.\ \mathsf{option\_case}\ 0\ (\boldsymbol{\lambda} x.\mathsf{Suc}\ (f\ x)))f_2)) \\
&\qquad\quad (\boldsymbol{\lambda} x.x) \\
&\quad =_\beta \ \mathsf{sum\_case}(\boldsymbol{\lambda}(x, y).\ f_1\ x + \mathsf{option\_case}\ 0\ (\boldsymbol{\lambda} x.\ \mathsf{Suc}\ (f_2\ x))\ y)\ (\boldsymbol{\lambda} x.x)
\end{aligned}$$

Definition 47 (Datatype size) *Suppose datatype $(\alpha_1, \ldots, \alpha_n)\tau$ has been defined, with constructors $\mathsf{C}_1, \ldots, \mathsf{C}_k$. Create function variables*

$$(f_1 : \alpha_1 \rightarrow \mathsf{num}), \ldots, (f_n : \alpha_n \rightarrow \mathsf{num}),$$

and let $\Theta \equiv \{\alpha_1 \mapsto f_1, \ldots, \alpha_n \mapsto f_n\}$. Extend Γ with a binding for $\mathsf{size\_}\tau$:

$$\Delta = \boldsymbol{\lambda} tyop.\ \mathtt{if}\ tyop = \tau\ \mathtt{then}\ size\_\tau\ f_1 \ldots f_n\ \mathtt{else}\ \Gamma(tyop).$$

Then define

$$\begin{aligned}
\mathsf{size\_}\tau\ f_1 \ldots f_n\ \mathsf{C}_i &\equiv 0, \quad \textit{if } \mathsf{C}_i \textit{ is nullary; otherwise,} \\
\mathsf{size\_}\tau\ f_1 \ldots f_n\ (\mathsf{C}\ (x_1 : \tau_1) \ldots (x_m : \tau_m)) &\equiv 1 + \textstyle\sum_{i=1}^{m} (|\tau_i|_{\Delta,\Theta}\ x_i).
\end{aligned}$$

□

Example. The size definition for the datatype of lists is:

$$\begin{array}{rcl}\mathsf{list\_size}\ f\ [\,] & \equiv & \mathsf{0} \\ \mathsf{list\_size}\ f\ (h :: t) & \equiv & 1 + f\ h + \mathsf{list\_size}\ f\ t.\end{array}$$

□

The size definition is intended to be useful when mechanizing termination proofs. A particularly simple and yet effective strategy when trying to prove termination of a function over a type τ is to attempt to prove that the recursive calls are in the relation $\mathsf{measure}(|\tau|_{\Gamma,\Theta})$ (where Θ is reset to say that the size functions generated from type variables just return $\mathsf{0}$). This approach to defining the size of datatype elements becomes particularly useful when dealing with functions defined by nested patterns.

Example. Consider the definition of a polymorphic function for removing a level of bracketing from a list:

$$\begin{array}{rcl}\mathsf{flat}\ [\,] & \equiv & [\,] \\ \mathsf{flat}\ ([\,] :: \ell) & \equiv & \mathsf{flat}\ \ell \\ \mathsf{flat}\ ((h :: t) :: \ell) & \equiv & h :: \mathsf{flat}\ (t :: \ell).\end{array}$$

To show that flat terminates, first ensure that Γ contains $\mathsf{list} \mapsto \mathsf{list\_size}$, and that $\Theta \equiv \{\alpha \mapsto \boldsymbol{\lambda} v : \alpha.\ \mathsf{0}\}$. Then

$$|\alpha\ \mathsf{list}\ \mathsf{list}|_{\Gamma,\Theta} = \mathsf{list\_size}\ (\mathsf{list\_size}\ (\boldsymbol{\lambda} v.\ \mathsf{0})),$$

and proving termination of flat can be done by showing that the recursive calls of flat lie in the relation $\mathsf{measure}\ (\mathsf{list\_size}\ (\mathsf{list\_size}\ (\boldsymbol{\lambda} v.\ \mathsf{0})))$, *i.e.*, (making the abbreviation $\mathcal{M} \equiv \mathsf{list\_size}\ (\boldsymbol{\lambda} v.\ \mathsf{0})$) by showing

$$\begin{array}{rcl}\mathsf{list\_size}\ \mathcal{M}\ \ell & < & \mathsf{list\_size}\ \mathcal{M}\ ([\,] :: \ell) \\ & = & 1 + \mathcal{M}\ [\,] + \mathsf{list\_size}\ \mathcal{M}\ \ell,\end{array}$$

and also

$$\begin{array}{rcl}\mathsf{list\_size}\ \mathcal{M}\ (t :: \ell) & < & \mathsf{list\_size}\ \mathcal{M}\ ((h :: t) :: \ell) \\ 1 + \mathcal{M}\ t + \mathsf{list\_size}\ \mathcal{M}\ \ell & < & 1 + \mathcal{M}\ (h :: t) + \mathsf{list\_size}\ \mathcal{M}\ \ell \\ & = & 1 + (1 + \mathsf{list\_size}\ (\boldsymbol{\lambda} v.\ \mathsf{0})\ (h :: t)) + \mathsf{list\_size}\ \mathcal{M}\ \ell \\ & = & 1 + (1 + (\boldsymbol{\lambda} v.\ \mathsf{0})\ h + \mathcal{M}\ t) + \mathsf{list\_size}\ \mathcal{M}\ \ell \\ & = & 1 + (1 + \mathcal{M}\ t) + \mathsf{list\_size}\ \mathcal{M}\ \ell.\end{array}$$

□

2.7 Contextual rewriting

The machinery of TFL makes essential use of a contextual term rewriter: it is the purpose of this section to explain how contextual rewriting can be implemented as a rule of inference, following the approach to implementing rewriters initiated in [82]. Our exegetical strategy will be one of progressive enrichment: we start with a simple unconditional rewriter and then show how it can be modified to accommodate conditional and then contextual rewriting.

An unconditional rewrite rule is an equational theorem $\vdash l = r$. A *basic match* of a rule $l = r$ to a term M yields a substitution θ for variables of l such that $\theta(l) = M$.[5] A *matcher* is a metalanguage function that takes a rule set R and a term M and attempts to find a rule in R that admits a basic match with M. When a matcher is applied to M, if there is a rule $l = r$ in R such that a basic match attempt succeeds with substitution θ, the theorem $\vdash M = \theta(r)$ is deduced; if no basic match can be found, the theorem $\vdash M = M$ is returned.[6]

All that is needed now is a way to move about the term and apply a matcher to subterms. It is simple enough to see how to do this algorithmically, but how are such steps to be logically justified? The results of replacements must be pasted together so that, after rewriting *term* to *term*$'$, perhaps at internal nodes, $\vdash term = term'$ can be concluded. The following two theorems transform equalities between subterms into equalities between their parents and thus justify gluing steps:

$$\begin{array}{rcl} \vdash (M = M') \land (N = N') & \supset & MN = M'N' \\ \vdash (M = M') & \supset & \boldsymbol{\lambda} x.\ M = \boldsymbol{\lambda} x.\ M' \end{array} \tag{2.2}$$

Because of the stark simplicity of the syntax of the HOL logic, these two rules suffice for implementing term traversal. An unconditional rewriter will take a rule set R and a term M and, in the standard case, repeatedly traverse the term making replacements until no more replacements can be made.

Conditional rewriting arises from the desire to rewrite with equations of the form

$$c_1 \land \ldots \land c_n \supset l = r.$$

A conditional matcher using rule set R makes the following steps at subterm M:

1. Matches M against R, giving $\vdash \theta(c_1) \land \ldots \land \theta(c_n) \supset (\theta(l) = \theta(r))$ if there is a rule $c_1 \land \ldots \land c_n \supset l = r$ in R such that $\theta(l) = M$.

2. Invokes a *condition prover* on $\theta(c_1) \land \ldots \land \theta(c_n)$. Notice that there can be free variables in $\theta(c_1) \land \ldots \land \theta(c_n)$ that do not occur in $\theta(l)$; these are

[5] The equality of $\theta(l)$ with M can be modulo a theory, *e.g.*, the rules of $\alpha\beta\eta$-conversion or other algebraic properties, but we won't explore this. We also won't discuss how matching substitutions are computed; for information on this, see [9].

[6] This is inefficient. Boulton [16] discusses an optimization that uses exception handling to circumvent rebuilding terms that are unchanged by rewrite steps.

sometimes called *existential* variables. In the proof of $\theta(c_1) \wedge \ldots \wedge \theta(c_n)$, the condition prover is allowed to find values for these variables, with the result that the condition prover either fails or returns $\vdash \sigma(\theta(c_1)) \wedge \ldots \wedge \sigma(\theta(c_n))$, where σ denotes the substitution binding the existential variables.

3. Performs *modus ponens* (with some matching to handle σ) with the results of steps (1) and (2), giving $\vdash \sigma(\theta(l)) = \sigma(\theta(r))$, which is just $\vdash \theta(l) = \sigma(\theta(r))$ because σ does not have any free variables of $\theta(l)$ in its domain.

If all these steps are successful, the theorem $\vdash M = \sigma(\theta(r))$ is returned; otherwise the result is $\vdash M = M$.

2.7.1 Making use of context

Thus, the differences between unconditional and conditional rewriting can be localized in the matcher, and the traversal strategy is not affected. However, the traversal strategies available so far do not allow the rewriter to take advantage of information that becomes dynamically available during traversal. For example, in traversing a conditional expression $\texttt{if}\ A\ \texttt{then}\ B\ \texttt{else}\ C$, it is valid and often helpful to assume A when rewriting B and also to assume $\neg A$ when rewriting C. A general and flexible way to represent such information is with a congruence rule, *e.g.*,

$$\begin{array}{l}(P = P') \wedge (P' \supset x = x') \wedge (\neg P' \supset y = y') \\ \quad\supset \\ \texttt{if}\ P\ \texttt{then}\ x\ \texttt{else}\ y = \texttt{if}\ P'\ \texttt{then}\ x'\ \texttt{else}\ y'.\end{array}$$

A congruence rule has the form of a standard conditional rewrite rule, but it is treated differently, since it supports a notion of *context*, which is a collection of theorems that are locally, or dynamically, valid. The following definition captures a simple[7] class of congruence rules:

Definition 48 (Simple congruence rule) *A* simple *congruence rule for function* f *has the following form:*

$$\vdash (P_1 \supset x_1 = y_1) \wedge \ldots \wedge (P_n \supset x_n = y_n) \supset (\mathsf{f}\, x_1 \ldots x_n) = (\mathsf{f}\, y_1 \ldots y_n),$$

where the P_i are optional.

To see how a congruence rule is interpreted when rewriting, assume that we are working in a context Γ and suppose that a subterm $\mathsf{f}\, M_1 \ldots M_n$ has matched a simple congruence theorem, yielding an instantiation

$$(P_1 \supset M_1 = y_1) \wedge \ldots \wedge (P_n \supset M_n = y_n) \supset \mathsf{f}\, M_1 \ldots M_n = \mathsf{f}\, y_1 \ldots y_n,$$

[7] A more general class allows any relation to be used, rather than just equality.

where all the y_i are free variables. We would like to replace $\mathsf{f}\ M_1 \ldots M_n$, but to do so we must prove the antecedents, and in order to do so, instantiate the existential variables. This is exactly the same situation faced in a standard conditional match. However, the manner of proof of the antecedents is different. For an antecedent $P_i \supset M_i = y_i$ the following actions are taken:

- P_i is assumed and added to Γ to get Γ'. Then a recursive invocation of the rewriter using R and Γ' is applied to M_i. This returns a theorem $\Delta \vdash M_i = M_i'$. If P_i was used somewhere in the proof, then $P_i \in \Delta$. P_i is now discharged from the assumptions, regardless of whether it occurs there or not, yielding $\Delta_i \vdash P_i \supset M_i = M_i'$.
- The existential variable y_i can now be instantiated to M_i' throughout the remainder of the instantiated congruence theorem.
- $P_i \supset M_i = M_i'$ is eliminated from the antecedents of the instantiated congruence theorem by *modus ponens*.

The ith step of this cycle leaves

$$\Delta_1 \cup \ldots \cup \Delta_i \vdash \left(\begin{array}{l} (P_{i+1} \supset M_{i+1} = y_{i+1}) \wedge \ldots \wedge (P_n \supset M_n = y_n) \\ \quad \supset \\ \mathsf{f}\ M_1 \ldots M_n = \mathsf{f}\ M_1' \ldots M_i'\ y_{i+1} \ldots y_n. \end{array} \right)$$

After n steps of this cycle, the right hand side of the conclusion has been fully instantiated, and the antecedents have all been eliminated and thus $\mathsf{f}\ M_1 \ldots M_n$ can be replaced by $\mathsf{f}\ M_1' \ldots M_n'$.

The increased power of contextual rewriting is due to the fact that the values of existential variables are found by a recursive invocation of the rewriter, using an augmented context. It is common for conditional rewriters to also invoke themselves recursively to solve conditions, so contextual rewriting can be thought of as a refinement of conditional rewriting made possible by the stylized form of congruence rules.

Higher order congruence rules

A *higher order congruence rule* is a congruence rule where the constant has a higher order type. For example, in

$$\begin{array}{l} (M = M') \wedge (\forall x.x = M' \supset f\ x = f'\ x) \\ \quad \supset \\ \mathsf{Let}\ f\ M = \mathsf{Let}\ f'\ M', \end{array} \tag{2.3}$$

the constant Let has type : $(\alpha \to \beta) \to \alpha \to \beta$. There are two new issues to address in this setting: the first is the higher order nature of the constant—this requires β-reduction and -abstraction steps; the second is that paired abstractions are often encountered—this requires the application of reduction steps for pairs. Thus an instance of the congruence rule can feature an antecedent $\forall\overline{x}.\ P\overline{x} \supset f\overline{x} = f'\overline{x}$, where elements of $\overline{x}$ may be tuples of variables.

For a concrete example of how the matcher can be upgraded to handle such rules, we will consider an application of (2.3). Suppose that a term

$$\texttt{let}\ (x_1, \ldots x_n) = M\ \texttt{in}\ N$$

is encountered during traversal, and assume that the rewriter has already accumulated a context Γ. The actual 'deep' syntax of the term is unravelled by the semantics of Let, and the way that paired lambda abstractions are encoded:

$$\begin{array}{rl} & \texttt{let}\ (x_1, \ldots x_n) = M\ \texttt{in}\ N \\ = & \mathsf{Let}(\boldsymbol{\lambda}(x_1, \ldots, x_n).\ N)\ M \\ = & \mathsf{Let}\ (\mathsf{pair\_case}(\boldsymbol{\lambda}x_1.\mathsf{pair\_case}(\boldsymbol{\lambda}x_2.\ldots \mathsf{pair\_case}(\boldsymbol{\lambda}x_{n-1}x_n.\ N)))\ M, \end{array}$$

and thus the instance of (2.3) we consider is

$$\begin{array}{l} M = y\ \wedge (\forall x.x = y \supset (\boldsymbol{\lambda}(x_1, \ldots, x_n).\ N)\ x = f'\ x) \\ \ \supset \\ \mathsf{Let}(\boldsymbol{\lambda}(x_1, \ldots, x_n).\ N)\ M = \mathsf{Let}\ f'\ y. \end{array} \tag{2.4}$$

The following steps are taken:

1. Rewrite M in Γ, obtaining $\Delta_1 \vdash M = M'$. In general, arbitrary new assumptions can appear as a result of being introduced in the proof of $\vdash M = M'$. The automatic extraction of termination conditions will be implemented by this mechanism. We now substitute M' for y throughout (2.4) and then remove the first antecedent, leaving the simplified instance:

$$\Delta_1 \vdash \left(\begin{array}{l} (\forall x.\ x = M' \supset (\boldsymbol{\lambda}(x_1, \ldots, x_n).\ N)\ x = f'\ x) \\ \ \supset \\ \mathsf{Let}(\boldsymbol{\lambda}(x_1, \ldots, x_n).\ N)M = \mathsf{Let}\ f'\ M'. \end{array} \right) \tag{2.5}$$

2. Consider the next antecedent

$$\forall x.\ x = M' \supset (\boldsymbol{\lambda}(x_1, \ldots, x_n).\ N)\ x = f'\ x.$$

Replace x throughout by the tuple $(x_1, \ldots, x_n)$ from the original term:

$$(x_1, \ldots, x_n) = M' \supset (\boldsymbol{\lambda}(x_1, \ldots, x_n).\ N)\ (x_1, \ldots, x_n) = f'\ (x_1, \ldots, x_n).$$

There is now a paired β-redex and it can be reduced, returning

$$\vdash (x_1, \ldots, x_n) = M' \supset N = f'\ (x_1, \ldots, x_n).$$

3. Augment the context Γ with $(x_1, \ldots, x_n) = M'$ and rewrite N. Then discharge $(x_1, \ldots, x_n) = M'$. This yields the theorem
$$\Delta_2 \vdash (x_1, \ldots, x_n) = M' \supset N = N'.$$

4. By paired β-expansion and transitivity of equality the following has now been proved:
$$\Delta_2 \vdash \left(\begin{array}{l} (x_1, \ldots, x_n) = M' \\ \quad \supset \\ (\boldsymbol{\lambda}(x_1, \ldots, x_n).\ N)\ (x_1, \ldots, x_n) = (\boldsymbol{\lambda}(x_1, \ldots, x_n).\ N')\ (x_1, \ldots, x_n) \end{array} \right)$$

5. Abstract $(x_1, \ldots, x_n)$ to x. This is surprisingly tedious; it amounts to checking that none of the variables in $(x_1, \ldots, x_n)$ occur free in subterms other than occurrences of $(x_1, \ldots, x_n)$.
$$\Delta_2 \vdash \forall x.\ x = M' \supset (\boldsymbol{\lambda}(x_1, \ldots, x_n).\ N)\ x = (\boldsymbol{\lambda}(x_1, \ldots, x_n).\ N')\ x. \quad (2.6)$$

6. Apply the substitution $f' \mapsto \boldsymbol{\lambda}(x_1, \ldots, x_n).\ N'$ to (2.5) and then use *modus ponens* to eliminate (2.6):
$$\Delta_1 \cup \Delta_2 \vdash \mathsf{Let}(\boldsymbol{\lambda}(x_1, \ldots, x_n).\ N)M = \mathsf{Let}(\boldsymbol{\lambda}(x_1, \ldots, x_n).\ N')M'.$$

Now the replacement can be performed. The important thing about these intricate machinations is that the bound variables introduced by a binding operator such as `let` must be freed before the body of the `let` is rewritten, and that the binding be treated as a unit. Thus when rewriting the second clause of Quicksort,

$$\begin{array}{lcl} \mathsf{fqsort}(ord, h :: t) & \equiv & \mathsf{let}\ (l_1, l_2) = \mathsf{part}(\boldsymbol{\lambda}y.\ ord\ y\ h, t, [\,], [\,]) \\ & & \mathsf{in} \\ & & \mathsf{fqsort}(ord, l_1)\ @\ [h]\ @\ \mathsf{fqsort}(ord, l_2), \end{array}$$

the binding $(l_1, l_2) = \mathsf{part}(\boldsymbol{\lambda}y.\ ord\ y\ h, t, [\,], [\,])$ is added to the context before proceeding to rewrite the body of the `let`. An alternative would be to omit the paired β-reduction steps, but then the congruence rule application would require not only that

$$\begin{array}{l} l_1 = \mathsf{fst}(\mathsf{part}(\boldsymbol{\lambda}y.\ ord\ y\ h, t, [\,], [\,])) \\ l_2 = \mathsf{snd}(\mathsf{part}(\boldsymbol{\lambda}y.\ ord\ y\ h, t, [\,], [\,])), \end{array}$$

be added to the context, but that l_1 and l_2 be expanded out in the body before rewriting proceeded, with the result that the sharing introduced by the `let` construct is lost. Avoiding such pitfalls is essential in the automatic extraction of understandable termination conditions.

Our approach also handles a common problem: when performing a higher order rewrite, variable names from the rewrite rule often get used instead of names from the subterm being rewritten. The above algorithm makes sure to instantiate the rewrite rule with names from the subterm before making the replacement. For our application, this is important, since we want to preserve the naming scheme used in the source program, especially when extracting termination conditions.

2.8 Summary

The basis enumerated in this chapter is relatively easy to provide. Building on basic inference rules and a simple principle of definition, we have shown how the required background theory of wellfoundedness, induction, and recursion can be derived. The wellfounded recursion theorem is the most technical part of the theorem proving effort; the elementary exposition we have given of its proof should enable the theorem to be proved with ease in other systems.

The most difficult part of the dynamics required by the TFL system is the implementation of a contextual rewriter adequate to the task of extracting termination conditions. Our rewriter has advanced features not available in other rewriters (handling instances of higher order congruence theorems with paired λ-abstractions, in particular). Again, we think that our presentation is clear enough to enable easy duplication of the required functionality. Furthermore, contextual rewriting is such an important proof tool that any serious verification system will have to embrace it eventually, and the explanation given here should clarify the important implementation issues.

Since the machinery of the next chapter uses the theorems, definition principles, and proof mechanisms from this chapter, *i.e.*, from the underlying proof system, any inconsistency that could result from using our machinery will already be derivable in the original system. Thus TFL is safe to use. That the functionality TFL provides is not trivial will be established in the next chapter.

Chapter 3

Mechanization

In this chapter we discuss the algorithms that apply the static and dynamic utilities described in the previous chapter. We start by showing how functions are defined when the termination relation is known. This is not an atomic act: important sub-algorithms involve a pattern-matching translation, termination condition extraction, and automated proof of a customized induction theorem. In subsequent sections, we show how to relax various requirements: first, the requirement that a termination relation be supplied at definition time is dropped; second, the requirement that the definition be a *closed* term is liberalized—this considerably extends the scope of TFL since such definitions can represent program schemes, which are commonly used to implement program transformation techniques.

3.1 Definitions with termination relations

The interface to TFL is centered about the act of defining a function: given some pattern-matching style recursion equations

$$\begin{array}{rcl} f(pat_1) & = & rhs_1 \\ & \vdots & \\ f(pat_n) & = & rhs_n \end{array}$$

and a termination relation R, the system steps through a definition algorithm:

1. The function description is translated into a functional $\boldsymbol{\lambda} f\ x.M$;
2. The basic principle of definition is applied to define $f \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M)$;
3. The recursion theorem (Theorem 24) is applied to the definition and then some basic simplifications are performed;
4. Termination conditions are extracted; and

5. An induction principle is derived by instantiating and manipulating the wellfounded induction theorem (Theorem 6).

Our design also allows further steps to be included as "post-processors" to step 4. In particular, the design allows for a prover that tries to establish the wellfoundedness of R as well as a prover that attempts to eliminate the termination conditions. Whatever the success of these postprocessors, at the end of all this manipulation a constant denoting the function has been defined and the following are returned:

- The original recursion equations, as a theorem of the host logic;
- An induction theorem, which serves as the main way to reason about the function; and
- Whatever termination conditions survive postprocessing. These remain as constraints attached to the recursion equations and the induction theorem.

Example. Consider the following program:

$$\mathsf{variant}(x,\ell) = \texttt{if}\ \mathsf{mem}\ x\ \ell\ \texttt{then}\ \mathsf{variant}(x+1,\ell)\ \texttt{else}\ x.$$

This function, or close relatives of it, is often found in symbolic systems since it is helpful for renaming bound variables in the course of substitution, or in "renaming variables apart" prior to unification. Trying to define variant formally has apparently given some authors difficulty, *e.g.*, Section 4 of [52]. The essential difficulty with variant is that it recurses on a larger argument and so it takes a moment of thought to understand exactly why it terminates. There are at least a couple of relations that explain why variant terminates (the second is from Matt Kaufmann).

$$\begin{array}{l}\mathsf{measure}\ \boldsymbol{\lambda}(x,\ell).\ \mathsf{length}(\mathsf{filter}(\boldsymbol{\lambda}y.\ x \leq y)\ell) \\ \mathsf{measure}\ \boldsymbol{\lambda}(x,\ell).\ (\mathsf{max}(\ell)+1) - x\end{array}$$

Using the first as our termination relation, the steps in the definition process elaborate as follows:

Translation. In this simple case, we just $\boldsymbol{\lambda}$-abstract the arguments and the recursively occurring function variable to get

$$\boldsymbol{\lambda} vary\ (x,\ell).\ \texttt{if}\ \mathsf{mem}\ x\ \ell\ \texttt{then}\ vary\ (x+1,\ell)\ \texttt{else}\ x.$$

The translation of definitions with more complex patterns on the left-hand side is described in Section 3.3.

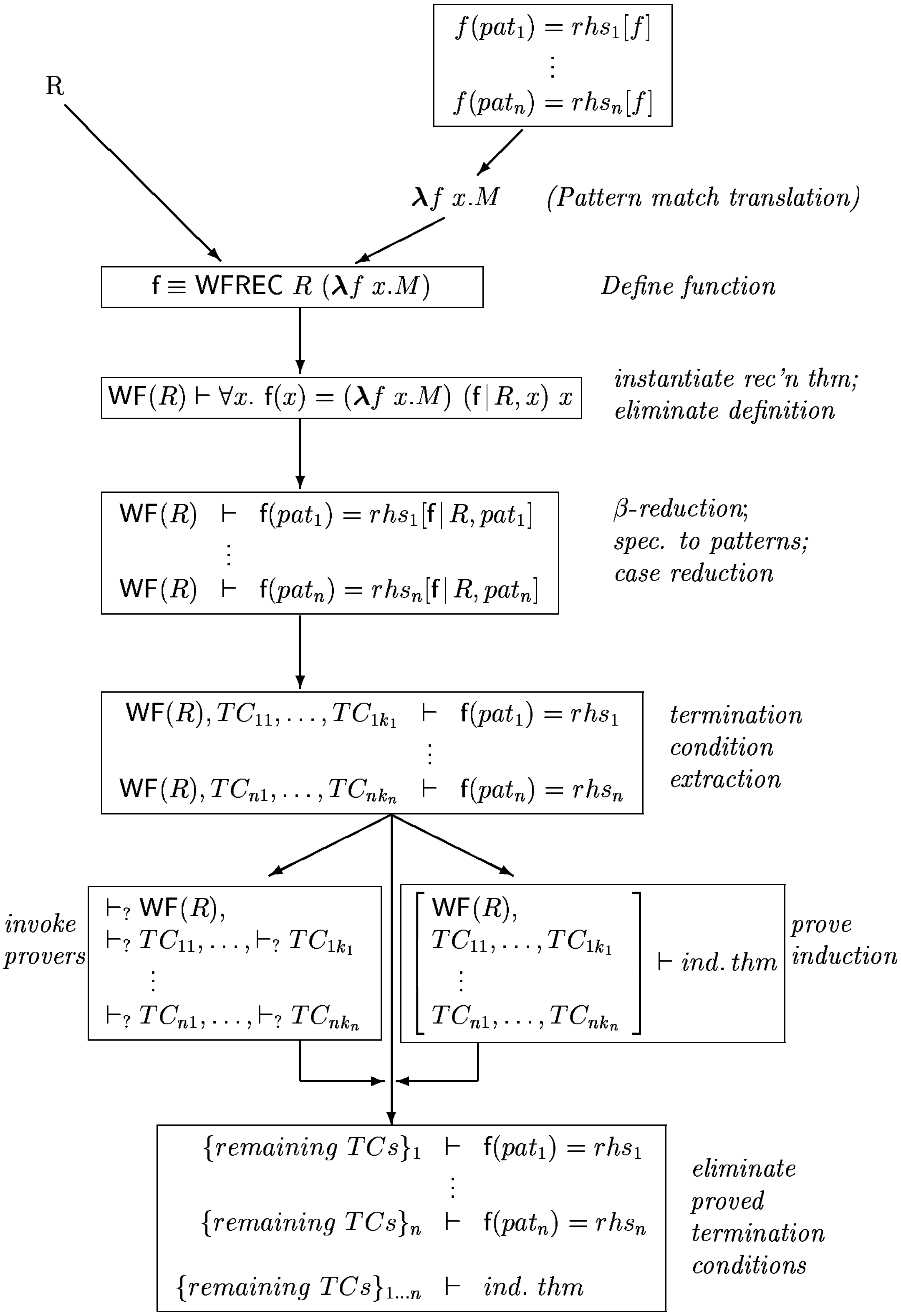

$f(pat_1) = rhs_1[f]$
$\vdots$
$f(pat_n) = rhs_n[f]$
R
$\boldsymbol{\lambda} f\ x.M$
(Pattern match translation)
$\mathsf{f} \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M)$
Define function
$\mathsf{WF}(R) \vdash \forall x.\ \mathsf{f}(x) = (\boldsymbol{\lambda} f\ x.M)\ (\mathsf{f}\,|\,R, x)\ x$
instantiate rec'n thm; eliminate definition
$\mathsf{WF}(R) \vdash \mathsf{f}(pat_1) = rhs_1[\mathsf{f}\,|\,R, pat_1]$
$\vdots$
$\mathsf{WF}(R) \vdash \mathsf{f}(pat_n) = rhs_n[\mathsf{f}\,|\,R, pat_n]$
β-reduction; spec. to patterns; case reduction
$\mathsf{WF}(R), TC_{11}, \ldots, TC_{1k_1} \vdash \mathsf{f}(pat_1) = rhs_1$
$\vdots$
$\mathsf{WF}(R), TC_{n1}, \ldots, TC_{nk_n} \vdash \mathsf{f}(pat_n) = rhs_n$
termination condition extraction
invoke provers
$\vdash_? \mathsf{WF}(R),$
$\vdash_? TC_{11}, \ldots, \vdash_? TC_{1k_1}$
$\vdots$
$\vdash_? TC_{n1}, \ldots, \vdash_? TC_{nk_n}$
$\begin{bmatrix} \mathsf{WF}(R), \\ TC_{11}, \ldots, TC_{1k_1} \\ \vdots \\ TC_{n1}, \ldots, TC_{nk_n} \end{bmatrix} \vdash \textit{ind. thm}$
prove induction
$\{\textit{remaining TCs}\}_1 \vdash \mathsf{f}(pat_1) = rhs_1$
$\vdots$
$\{\textit{remaining TCs}\}_n \vdash \mathsf{f}(pat_n) = rhs_n$
$\{\textit{remaining TCs}\}_{1\ldots n} \vdash \textit{ind. thm}$
eliminate proved termination conditions

Figure 3.1: The process of definition

Definition. Make the definition

$$\begin{array}{lcl}\mathsf{variant} & \equiv & \mathsf{WFREC}\ (\mathsf{measure}\ \boldsymbol{\lambda}(x,\ell).\ \mathsf{length}(\mathsf{filter}(\boldsymbol{\lambda}y.\ x \leq y)\ell)) \\ & & \quad (\boldsymbol{\lambda} vary\ (x,\ell).\ \texttt{if}\ \mathsf{mem}\ x\ \ell\ \texttt{then}\ vary\ (x+1,\ell)\ \texttt{else}\ x).\end{array}$$

Either of the principles of definition from Section 2.1.2 has no trouble in accepting this formula.

Apply recursion theorem. By application of *modus ponens* with Theorem 24, we get the following theorem (the wellfoundedness condition has been shunted to the assumptions):

$$\begin{array}{ll} & \mathsf{WF}(\mathsf{measure}\ \boldsymbol{\lambda}(x,\ell).\ \mathsf{length}(\mathsf{filter}(\boldsymbol{\lambda}y.\ x \leq y)\ell)) \\ \vdash & \mathsf{variant}(x,\ell) = \\ & \quad \text{if } (\mathsf{mem}\ x\ \ell) \text{ then} \\ & \quad (\mathsf{variant} \mid \mathsf{measure}(\boldsymbol{\lambda}(x,\ell).\ \mathsf{length}(\mathsf{filter}(\boldsymbol{\lambda}y.\ x \leq y)\ell)), (x,\ell)) \\ & \quad\quad (x+1,\ell) \\ & \quad \text{else } x.\end{array}$$

Extract termination conditions. The original recursive call $\mathsf{variant}(x+1,\ell)$ has now been converted into a constrained form:

$$(\mathsf{variant} \mid \mathsf{measure}(\boldsymbol{\lambda}(x,\ell).\ \mathsf{length}(\mathsf{filter}(\boldsymbol{\lambda}y.\ x \leq y)\ell)), (x,\ell))(x+1,\ell).$$

The process of extraction trades such constrained occurrences for the original occurrences, but this trade can only happen when the *termination condition* can be proved. Our machinery enforces this requirement by logic; the replacement is achieved by performing an inference with Theorem 14. The details of extraction are developed in Section 3.2, where it is described how termination conditions are captured and brought out to the top-level of the definition, so that automatic methods can have easy access in their attempts to prove them. If these methods fail, the termination conditions are then also easily accessible for the user to attempt. An important fact is that the extraction algorithm is implemented by inference steps. The result of extraction is therefore a theorem:

$$\begin{array}{l}\left[\begin{array}{l}\mathsf{WF}(\mathsf{measure}\ \boldsymbol{\lambda}(x,\ell).\ \mathsf{length}(\mathsf{filter}(\boldsymbol{\lambda}y.\ x \leq y)\ell)), \\ \mathsf{mem}\ x\ \ell \supset \mathsf{length}(\mathsf{filter}(\boldsymbol{\lambda}y.\ x+1 \leq y)\ell) < \mathsf{length}(\mathsf{filter}(\boldsymbol{\lambda}y.\ x \leq y)\ell)\end{array}\right] \\ \quad \vdash \\ \mathsf{variant}(x,\ell) = \texttt{if}\ \mathsf{mem}\ x\ \ell\ \texttt{then}\ \mathsf{variant}(x+1,\ell)\ \texttt{else}\ x.\end{array}$$

Prove induction theorem. An induction theorem is proved, using the results of termination condition extraction:

$$\left[\begin{array}{l}\mathsf{WF}(\mathsf{measure}\ \boldsymbol{\lambda}(x,\ell).\ \mathsf{length}(\mathsf{filter}(\boldsymbol{\lambda} y.\ x \leq y)\ell)),\\ \forall x\,\ell.\ \mathsf{mem}\ x\ \ell\\ \qquad \supset \mathsf{length}(\mathsf{filter}(\boldsymbol{\lambda} y.\ x+1 \leq y)\ell) < \mathsf{length}(\mathsf{filter}(\boldsymbol{\lambda} y.\ x \leq y)\ell)\end{array}\right]$$

$$\vdash$$

$$\forall P.\ (\forall x\ \ell.\ (\mathsf{mem}\ x\ \ell \supset P(\mathsf{Suc}\ x, \ell)) \supset P(x,\ell)) \supset \forall v\ v_1.\ P(v, v_1).$$

The details of this derivation are discussed in Section 3.4.

Postprocessing. The wellfoundedness prover easily proves the wellfoundedness hypothesis and eliminates it from the theorem. However, the termination condition solver fails on the remaining hypothesis, leaving it for the user to prove. It can be proved by induction on ℓ, using the following lemma:

$$(\forall x.\ P(x) \supset Q(x)) \supset \forall \ell.\ \mathsf{length}(\mathsf{filter}\ P\ \ell) \leq \mathsf{length}(\mathsf{filter}\ Q\ \ell).$$

With that accomplished, the recursion equations and the induction theorem can now be freed of termination constraints.

□ **End of example.**

Thus the two most important facts about the function, the recursion equations and the induction theorem, have been generated through an unbroken chain of inference. As can be seen, the work is mostly straightforward. Human creativity, or automated reasoning, is required to find the right relation R, to prove the wellfoundedness of R, and to eliminate any remaining termination conditions.

In the remainder of this chapter, we will discuss the steps taken by the process of definition in detail. We begin by examining the way in which contextual rewriting is used to extract termination conditions. Next, we discuss the pattern match translation that TFL uses. After that, we examine the details of how the induction theorem for a function is proved. This completes the discussion of how definitions with relations are built. The subsequent section deals with the *relationless* definition of recursive functions. Building on the mechanism for relationless definitions, the next section illustrates how schematic definitions can be dealt with.

3.2 Extracting termination conditions

Suppose a function $f(z) = M$ with recursive calls $f(z_1) \ldots f(z_n)$ occurring in M is to be defined. Let R be the termination relation. In the instantiated recursion theorem, this gives rise to the constrained occurrences $(f \mid R, z)z_1 \ldots (f \mid R, z)z_n$

and the termination conditions $(R\ z_1\ z) \ldots (R\ z_n\ z)$. In general, each condition will only be provable in the context existing at the recursive call site. To illustrate, in the variant example, the termination condition

$$\mathsf{length}(\mathsf{filter}(\boldsymbol{\lambda} y.\ x+1 \leq y)\ \ell) < \mathsf{length}(\mathsf{filter}(\boldsymbol{\lambda} y.\ x \leq y)\ \ell)$$

is only provable when the context of the recursive call—$\mathsf{mem}\ x\ \ell$—is taken into account. The *context* $\Gamma(N)$ of a subterm N of M is a collection of facts that are true because of where N occurs in M. The *full termination condition* of recursive call $f(z_i)$ in context $\Gamma(z_i) = [h_1, \ldots, h_m]$ will be the formula $h_1 \wedge \ldots \wedge h_m \supset R\ z_i\ z$. If the termination conditions of a function are provable, then the function is total.

However, what does *context* really mean? Its usage in the above paragraph is merely intuitive. Our answer comes from noting that, using congruence rules, the context of every subterm, including the recursive call sites, can be automatically tracked by a rewrite engine of the kind described in Section 2.7. The implementation of TFL extracts termination conditions by using such a rewriter to rewrite the instantiated recursion theorem coming from step (3) of the definition process by the single conditional rewrite rule

$$R\ x\ y \supset (f \mid R, y)x = f\ x.$$

In searching for matches to this rule, the rewriter is essentially searching the instantiated recursion theorem for constrained recursive calls. The rewriting process continues until one full traversal of the theorem finds no constrained recursive call. As the rewriter makes its traversal, it gathers and discards context according to its stock of congruence rules. Thus, the set of congruence rules *defines* the notion of context. This approach is general and very flexible: by allowing the set of congruence rules to be user-extensible, new notions of context can be installed as they arise in formalizations; the only requirement is that the intended notion of context be capturable in a congruence rule. So far, we have not found this to be a limitation.

A. $h_1, \ldots, h_m \vdash h_1 \wedge \ldots \wedge h_m$	$\wedge$-intro$*$ $(\Gamma(z_i))$
B. $\forall(h_1 \wedge \ldots \wedge h_m \supset R\ z_i\ z) \vdash \forall(h_1 \wedge \ldots \wedge h_m \supset R\ z_i\ z)$	Assume
C. $\forall(h_1 \wedge \ldots \wedge h_m \supset R\ z_i\ z) \vdash h_1 \wedge \ldots \wedge h_m \supset R\ z_i\ z$	$\forall$-elim$*$ B
D. $h_1, \ldots, h_m, \forall(h_1 \wedge \ldots \wedge h_m \supset R\ z_i\ z) \vdash R\ z_i\ z$	$\supset$-elim $C\ A$
E. $h_1, \ldots, h_m, \forall(h_1 \wedge \ldots \wedge h_m \supset R\ z_i\ z) \vdash (f \mid R, z)z_i = f(z_i)$	$\supset$-elim (14) D

Figure 3.2: Capturing termination conditions by proof

It is not enough merely to accumulate context and rewrite recursive call sites; the termination condition needs to be *captured* at each call site. This can neatly be accomplished via inference steps. At a constrained recursive call $(f \mid R, z)z_i$ in context $\Gamma(z_i)$ (having elements $h_1, \ldots, h_m$), the little proof in Figure 3.2 is performed by the condition prover component of the rewriter, thus allowing the replacement of $(f \mid R, z)z_i$ by $f\ z_i$. (The initial step $\wedge$-intro$*$ $(\Gamma(z_i))$ assumes each element in $\Gamma(z_i)$ and then makes a sequence of $\wedge$-intro steps).

The proof of Figure 3.2 always succeeds (it is easy to check that each step in it cannot fail). After the proof is performed, the rewriter performs the replacement. Since each replacement removes one constrained recursive call $(f \mid R, z)z_i$ from the term, the rewriting process terminates. In situations with no nested functions, a single traversal of the term is needed to capture all termination conditions. In situations with nested functions, repeated traversals may be necessary.

In the proof, the termination condition is generalized to $\forall(h_1 \wedge \ldots \wedge h_m \supset R\ z_i\ z)$ and stored on the assumption list. As rewriting 'unwinds', each of the context elements $h_1, \ldots, h_m$ will be removed at the point it was added to the context. However, the termination condition will not be removed since the $h_1, \ldots, h_m$ are proper subexpressions of it. The end result, after rewriting finishes, is the theorem

$$\forall(\Gamma(z_1) \supset R\ z_1 z), \ldots, \forall(\Gamma(z_n) \supset R\ z_n z) \vdash f(z) = M$$

in which the termination conditions have been fully separated from the recursion equations. It is now simple for tools to either automatically attempt to solve them or to present them as goals for the user.

Subtle point. In step B of the proof in Figure (3.2) there is a choice about which variables to universally quantify. If all are quantified, then the termination condition becomes effectively inert; bizarre constraints (arising from accidental coincidence of free variables) with other formulae encountered in subsequent proof steps can never arise. This is a good thing. However, in the case of nested recursion, fully quantifying each termination condition can result in it becoming impossible to use the definition of the function before all termination conditions are proved (unless elaborate steps are taken). This is a bad thing, because termination proofs of nested recursive functions generally require unrolling the function definition.

Therefore, in step B we quantify the termination condition as little as possible: only the local variables introduced on the right hand side, *e.g.*, those introduced by a let construct, are generalized. This allows recursion equations to be unrolled at a specific instance, provided the termination conditions can be proved at that instance. We will return to this point when discussing the derivation of induction schemes in Section 3.4, and induction schemes for nested functions in Section 4.1.1.

□

3.3 Pattern-matching

Pattern-matching is a convenient and standard description technique for functional programs. A simple example of a function described by pattern-matching is gcd:

$$\begin{array}{rcl}
\mathsf{gcd}(0, y) & \equiv & y \\
\mathsf{gcd}(\mathsf{Suc}\ x, 0) & \equiv & \mathsf{Suc}\ x \\
\mathsf{gcd}(\mathsf{Suc}\ x, \mathsf{Suc}\ y) & \equiv & \mathtt{if}\ y \leq x\ \mathtt{then}\ \mathsf{gcd}(x - y,\ \mathsf{Suc}\ y) \\
& & \qquad\qquad\ \ \mathtt{else}\ \mathsf{gcd}(\mathsf{Suc}\ x,\ y - x)
\end{array}$$

Compilation of efficient code for pattern-matching is very important for speedy execution of functional programs. In this section we show that such compilation techniques can also be adapted to the problem of defining logical functions written with patterns.

A straightforward operational interpretation of the rules for gcd is that to find the value of gcd at a pair of natural numbers, one must match the patterns of the left hand sides in some order until a match θ for the variables in the pattern is found, then apply θ to the corresponding right hand side, and continue by evaluating the instantiated right hand side. There are several problems with this interpretation: first, the meaning of gcd is operational; second, the algorithm is inefficient, since it can happen that a lot of redundant matching occurs in the search for a successful match; third, the interpretation says nothing about whether the matches are total, *i.e.*, cover all possible inputs of type $\mathsf{num} * \mathsf{num}$; fourth, the interpretation says nothing about whether the description is even a function, *i.e.*, single valued!

Augustsson [8], motivated by the second consideration, invented an effective divide-and-conquer algorithm that considered sets of patterns. More importantly for our logical setting, his algorithm also helps deal with the other problems. The first problem is solved since Augustsson's method translates the program into a nested case expression, which is something that has a straightforward logical interpretation using the case definitions for the concrete types of Section 2.2. The third problem is solved because a simple restriction of the algorithm implicitly checks that the patterns are complete. The fourth problem is solved because the algorithm uses a priority scheme to decide which pattern to select in the case of overlapping patterns (the potential source for multiple valuations of the same input).

Our algorithm is not exactly that of Augustsson: he deals with overlapping patterns by generating code that may backtrack at runtime, in order to have compact code. Lacking a concept of runtime in our logical setting, we are unable to use that solution. Therefore, we divide our development into an algorithm that works for complete and non-overlapping patterns (a simple restriction of Augustsson's algorithm), and a pre-processing extension that deals with incomplete and overlapping patterns. The translation of overlapping patterns implements the usual semantics of pattern matching in functional languages, in which the first successful match in a top to bottom scan is taken.

Our pattern language is generated by the following grammar:

$$pat = v \mid \mathsf{C}\ pat_1 \ldots pat_n$$

where v is a variable, and $\mathsf{C} : \tau_1 \to \ldots \to \tau_n \to \tau_{n+1}$ is a constructor obtained by a datatype definition of the kind described in Section 2.2. A further restriction—linearity—is also required: no variable may occur more than once in a pattern. This pattern language lacks several common features:

- So-called '*as*' patterns. These provide the ability to attach names to parts of the pattern. For example,

  $$f\ (\ell\ \mathsf{as}\ h :: t) \equiv rhs$$

  allows ℓ to be used in rhs to stand for the entire list being matched. It is not clear what the semantics of as is; however, it can be translated into uses of `let` in rhs, *e.g.*, the example could be translated to $f\ (h :: t) \equiv$ `let` $\ell = h :: t$ `in` rhs. We have not yet pursued this. The thesis of Cornes [27] discusses a treatment of as patterns in the Coq proof system and there are also constructive interpretations of as for lambda calculi with built-in patterns [57].

- *Wildcard* patterns. These do not form part of our translation *per se*, but must be accommodated as a part of parsing the original term, something that TFL leaves to the underlying proof system.

We first discuss the basic pattern matching algorithm and then explain the extensions for overlapping and incomplete patterns.

3.3.1 Translation of pattern-matching

Assume we are given the program description

$$\begin{array}{rcl} f(pat_1) & \equiv & rhs_1 \\ & \vdots & \\ f(pat_n) & \equiv & rhs_n. \end{array}$$

The algorithm $\mathcal{O}$, given in Figure 3.3, and expressed in pseudo-ML, will translate this into a nested case expression. $\mathcal{O}$ takes two arguments: a stack of variables and a *row* matrix. A row corresponds to a clause in the function definition: initially, it is just the list of patterns as given in the clause, each paired with its right-hand side. In contrast to the operational interpretation given above, which proceeded row by row, $\mathcal{O}$ goes from left to right, column by column.

The variable stack represents the environment which gets built up as matching progresses: there is one variable for each of the data objects remaining to be matched. There are several simple invariants for $\mathcal{O}$:

- All rows are of equal length (the matrix is a rectangle).

- The elements in a column all have the same type.

- The length of the variable stack is the width of the matrix.

- The ith variable in the variable stack has the same type as the ith column.

As mentioned, $\mathcal{O}$ proceeds by examining the first column. There are three allowed cases:

Variable Each element in the column is a variable. Recall that the notation $[v \mapsto z]M$ denotes the substitution of z for v throughout the term M. This translation-time substitution essentially performs α-renaming, which itself sets up parallel substitution at "runtime".

Constructor Each element in the column is the application of a constructor for type τ. The problem now splits into n subproblems, one for each constructor $\mathsf{C}_1, \ldots, \mathsf{C}_n$ of τ. Since each constructor can have repeated occurrences (and applications of a constructor C_i need not be grouped in consecutive rows), there is a stage of partitioning the rows into n groups of size $k_1 \ldots k_n$. (In $\mathcal{O}$, how this grouping is accomplished is not important. However, in the extension to overlapping patterns, the maintenance of order will be of crucial importance.) After partitioning, a row $(\mathsf{C}(\overline{p}) :: \mathit{pats}, \mathit{rhs})$ has its lead constructor discarded, resulting in a row expression $(\overline{p} \mathbin{@} \mathit{pats}, \mathit{rhs})$. In subproblem i, supposing the constructor C_i has type $\tau_1 \to \ldots \to \tau_j \to \tau$, j new variables $v_1 : \tau_1, \ldots, v_j : \tau_j$ are pushed onto the stack. This vector of variables is denoted $\mathcal{V}_i$. The results of invoking $\mathcal{O}$ on the subproblems are combined into a case expression that analyzes the head of the variable stack.

End The patterns have been exhausted and the stack is empty, leaving a single right hand side, which is returned.

The result of invoking $\mathcal{O}$ is used to build a *functional* by abstracting f and a variable z (which must not be free in any of $\mathit{rhs}_1, \ldots, \mathit{rhs}_n$):

$$\boldsymbol{\lambda} f\ z.\ \mathcal{O}\left(\begin{array}{c} [z], \\ \left[\begin{array}{c} ([\mathit{pat}_1], \mathit{rhs}_1), \\ \vdots \\ ([\mathit{pat}_n], \mathit{rhs}_n) \end{array}\right] \end{array}\right).$$

Variable

$$\boldsymbol{\mathcal{O}}\left(\begin{array}{c} z :: stack, \\ \left[\begin{array}{l}(v_1 :: pats_1, rhs_1),\\ \quad\vdots \\ (v_n :: pats_n, rhs_n)\end{array}\right]\end{array}\right) = \boldsymbol{\mathcal{O}}\left(\begin{array}{c} stack, \\ \left[\begin{array}{l}(pats_1, [v_1 \mapsto z]rhs_1),\\ \quad\vdots \\ (pats_n, [v_n \mapsto z]rhs_n)\end{array}\right]\end{array}\right)$$

Constructor

$$\boldsymbol{\mathcal{O}}\left(\begin{array}{c} z :: stack, \\ \left[\begin{array}{l}\mathsf{C}_1\overline{p_{11}} :: pats_{11}, rhs_{11}\\ \quad\vdots \\ \mathsf{C}_1\overline{p_{1k_1}} :: pats_{1k_1}, rhs_{1k_1}\\ \quad\cdots \\ \mathsf{C}_n\overline{p_{n1}} :: pats_{n1}, rhs_{n1}\\ \quad\vdots \\ \mathsf{C}_n\overline{p_{nk_n}} :: pats_{nk_n}, rhs_{nk_n}\end{array}\right]\end{array}\right)$$

$$=$$

$$\begin{array}{l}
\text{let } M_1 = \boldsymbol{\mathcal{O}}\left(\begin{array}{c} \mathcal{V}_1 @ stack, \\ \left[\begin{array}{l}\overline{p_{11}}@pats_{11}, rhs_{11}\\ \quad\vdots \\ \overline{p_{1k_1}}@pats_{1k_1}, rhs_{1k_1}\end{array}\right]\end{array}\right)\\
\qquad\vdots \\
\qquad M_n = \boldsymbol{\mathcal{O}}\left(\begin{array}{c} \mathcal{V}_n @ stack, \\ \left[\begin{array}{l}\overline{p_{n1}}@pats_{n1}, rhs_{n1}\\ \quad\vdots \\ \overline{p_{nk_n}}@pats_{nk_n}, rhs_{nk_n}\end{array}\right]\end{array}\right)\\
\texttt{in}\\
\mathsf{case\_ty}\ (\boldsymbol{\Lambda}\mathcal{V}_1.\ M_1)\ldots(\boldsymbol{\Lambda}\mathcal{V}_n.\ M_n)\ z
\end{array}$$

End

$$\boldsymbol{\mathcal{O}}([\,], [([\,], rhs)]) = rhs$$

Figure 3.3: Basic pattern matching algorithm

Example. The following is the translation of the gcd program:

$$
\begin{array}{l}
\boldsymbol{\lambda} gcd\ z.\ \mathsf{pair\_case} \\
\quad (\boldsymbol{\lambda} v\ v_1.\ \mathsf{num\_case}\ v_1 \\
\qquad (\boldsymbol{\lambda} v_2.\ \mathsf{num\_case}(\mathsf{Suc}\ v_2) \\
\qquad\quad (\boldsymbol{\lambda} v_3.\ \text{if } (v_3 \leq v_2) \\
\qquad\qquad \text{then } gcd(v_2 - v_3, \mathsf{Suc}\ v_3) \\
\qquad\qquad \text{else } gcd(\mathsf{Suc}\ v_2, v_3 - v_2))\ v_1)\ v)\ z.
\end{array}
$$

3.3.2 Incomplete and overlapping patterns

An *incomplete* set of patterns will fail to match at least one data object. An *overlapping* set of patterns has the unwanted feature that there are data objects that more than one pattern in the set can match. Thus, the following description

$$
\begin{array}{rcl}
f(x, \mathsf{0}) & \equiv & \mathsf{True} \\
f(\mathsf{0}, x) & \equiv & \mathsf{False}
\end{array}
$$

overlaps at $(\mathsf{0}, \mathsf{0})$ (should the result be True or False?) and is incomplete at $(\mathsf{Suc}\ a, \mathsf{Suc}\ b)$, for any a and b. Overlapping patterns are often used to succinctly express complex patterns in data, so it is interesting to see how algorithm $\boldsymbol{\mathcal{O}}$ can be extended to handle them. Omitted patterns describe partial functions, and in a logic of total functions, such can not really be supported accurately, but it is easy to extend $\boldsymbol{\mathcal{O}}$ to handle them as well.

Remark. There are non-overlapping pattern sets that are not dealt with by algorithm $\boldsymbol{\mathcal{O}}$, as the following example from Wadler [54] shows:

$$
\begin{array}{rcl}
\mathsf{diagonal}(x, \mathsf{True}, \mathsf{False}) & \equiv & 1 \\
\mathsf{diagonal}(\mathsf{False}, y, \mathsf{True}) & \equiv & 2 \\
\mathsf{diagonal}(\mathsf{True}, \mathsf{False}, z) & \equiv & 3.
\end{array}
$$

The diagonal function has no overlaps; however, the pattern set has to be fully analyzed to determine this fact. For this reason, Wadler suggests the term *uniform* to denote the set of patterns that algorithm $\boldsymbol{\mathcal{O}}$ handles. □

Incomplete patterns

In the **Constructor** rule, it may happen that an application $\mathsf{C}_i\ \overline{p}$ of some constructor C_i of the column type is missing from the lead column. In this situation, the pattern set is incomplete. We complete it by creating a new row for each of the missing constructors $\mathsf{C}_i, \ldots, \mathsf{C}_k$; these rows are then added to the row matrix. The right hand side of each new row is just Arb, instantiated to σ, the range type of the function being translated.

Complete

$$\begin{pmatrix} p_1 :: pats_1, rhs_1 \\ \vdots \\ p_n :: pats_n, rhs_n \end{pmatrix} = \begin{pmatrix} p_1 :: pats_1, rhs_1 \\ \vdots \\ p_n :: pats_n, rhs_n \\ \left[C_i\ \overline{y_i}, u_1 : \tau_1, \ldots, u_j : \tau_j\right], \mathsf{Arb} : \sigma \\ \vdots \\ \left[C_k\ \overline{y_k}, v_1 : \tau_1, \ldots, v_j : \tau_j\right], \mathsf{Arb} : \sigma \end{pmatrix}$$

Each new row has new variables put "everywhere": $u_1, \ldots, u_j, \ldots, v_1, \ldots, v_j$ are new variables bearing the types of the respective columns $1, \ldots, j$. Also, vectors of new variables $\overline{y_i}, \ldots, \overline{y_k}$ must be created to be the arguments of the constructor applications at the head of the column.

Overlapping patterns

The remaining case when the rules of $\boldsymbol{\mathcal{O}}$ fail to apply is when the lead column is a mixture of variables and constructor applications. This divides the row matrix into alternating blocks headed by constructor applications and variables:

$$\begin{array}{rl} \vdots & \\ \text{constructor block} & \left\{\begin{array}{l} \quad\vdots \\ \mathsf{C}\ \overline{p_j} :: pats_j, rhs_j \end{array}\right. \\ \text{variable block} & \left\{\begin{array}{l} v_{j+1} :: pats_{j+1}, rhs_{j+1} \\ \quad\vdots \\ v_k :: pats_k, rhs_k \end{array}\right. \\ \text{constructor block} & \left\{\begin{array}{l} \mathsf{C}\ \overline{p_{k+1}} :: pats_{k+1}, rhs_{k+1} \\ \quad\vdots \end{array}\right. \\ \vdots & \end{array}$$

Our desire is to transform such a situation into an equivalent problem where all the entries in the lead column are constructors. Consider row $j+1$. Under the operational interpretation, if an attempt to match by the previous constructor block has failed, the intent is to successfully match the leading data object with v_{j+1}, and then attempt to match the remaining data objects against $pats_{j+1}$. To implement this with constructors, it suffices to replace row $j+1$ by n copies of itself, one for each constructor in τ (assuming that τ has n constructors). In copy m, v_{j+1} is replaced by an application of $\mathsf{C}_m : \tau_1 \rightarrow \ldots \rightarrow \tau_q \rightarrow \tau$ to variables $y_1 : \tau_1 \ldots y_q : \tau_q$, where each variable has not been used yet in the algorithm, and does not occur in the original equations. The expansion $v_{j+1} \mapsto \mathsf{C}_m\ y_1\ \ldots y_q$ is also applied to rhs_{j+1}. If matching row $j+1$ should succeed operationally, then exactly one of the n new rows should succeed in matching. In effect, we have replaced a potential application of the **Variable** rule by a disjunction of applications of the **Constructor** rule. This motivates the following, where we focus in on only one block alternation (a block alternation involves moving either from a variable block to a constructor block, or *vice versa*):

VarElim

$$\begin{array}{ccc}
\vdots & & \vdots \\
\left(\begin{array}{l}
\mathsf{C}\ \overline{p_j} :: pats_j, rhs_j \\
v_{j+1} :: pats_{j+1}, rhs_{j+1} \\
\quad\vdots \\
v_k :: pats_k, rhs_k
\end{array}\right)
& = &
\left(\begin{array}{l}
\mathsf{C}\ \overline{p_j} :: pats_j, rhs_j \\
\mathsf{C}_1\ \overline{y_1} :: pats_{j+1}, [v_{j+1} \mapsto \mathsf{C}_1\ \overline{y_1}]rhs_{j+1} \\
\ldots \\
\mathsf{C}_n\ \overline{y_n} :: pats_{j+1}, [v_{j+1} \mapsto \mathsf{C}_n\ \overline{y_n}]rhs_{j+1} \\
\quad\vdots \\
\mathsf{C}_1\ \overline{u_1} :: pats_k, [v_k \mapsto \mathsf{C}_1\ \overline{u_1}]rhs_k \\
\ldots \\
\mathsf{C}_n\ \overline{u_n} :: pats_k, [v_k \mapsto \mathsf{C}_n\ \overline{u_n}]rhs_k
\end{array}\right) \\
\vdots & & \vdots
\end{array}$$

A mixed first column is, after executing the **VarElim** step, acceptable input for the **Constructor** rule. However, the operation of **Constructor** must be constrained in the partitioning step: when grouping all the rows headed by constructor C_i, care must be taken that the relative order of the source rows be preserved. Otherwise, a row introduced by an application of **VarElim** might be placed ahead of a row that initially occurred above it, thus violating the top-to-bottom operational strategy.

The **End** rule must also be changed when dealing with non-uniform pattern sets. As before, the search for a match has come to an end: the patterns have been exhausted and the variable stack is empty. The new aspect is that now there is a list of right hand sides to choose from, each representing a different right hand side stemming from an overlap. This list is ordered, by the discipline exercised in the partitioning phase of the **Constructor** rule, so picking the first element of it delivers the first rule that matches in top-to-bottom order.

The enhanced pattern matching algorithm $\boldsymbol{\mathcal{P}}$ incorporating these decisions is represented in Figure 3.4. $\boldsymbol{\mathcal{P}}$ can be seen as an optimized version of a two-pass algorithm that first expands all incomplete and overlapping patterns in all rows, and then invokes $\boldsymbol{\mathcal{O}}$ (suitably modified so as to maintain order in the partitioning phase of the **Constructor** step, and to handle multiple results in the **End** step).

Variable

$$\mathcal{P}\left(\begin{array}{l} \quad z :: stack, \\ \left[\begin{array}{l}(v_1 :: pats_1, rhs_1),\\ \quad\vdots \\ (v_n :: pats_n, rhs_n)\end{array}\right]\end{array}\right) = \mathcal{P}\left(\begin{array}{l} \quad stack, \\ \left[\begin{array}{l}(pats_1, [v_1 \mapsto z]rhs_1),\\ \quad\vdots\\ (pats_n, [v_n \mapsto z]rhs_n)\end{array}\right]\end{array}\right)$$

Constructor

$$\mathcal{P}\left(\begin{array}{l} \quad z :: stack, \\ \left[\begin{array}{l}p_1 :: pats_1, rhs_1\\ \quad\vdots \\ p_n :: pats_n, rhs_n\end{array}\right]\end{array}\right)$$

$$=$$

$$\begin{array}{lrcl}
\mathtt{let} & \left[\begin{array}{l}\mathsf{C}_1\overline{p_{11}} :: pats_{11}, rhs_{11}\\ \quad\vdots\\ \mathsf{C}_1\overline{p_{1k_1}} :: pats_{1k_1}, rhs_{1k_1}\\ \quad\cdots\\ \mathsf{C}_n\overline{p_{n1}} :: pats_{n1}, rhs_{n1}\\ \quad\vdots\\ \mathsf{C}_n\overline{p_{nk_n}} :: pats_{nk_n}, rhs_{nk_n}\end{array}\right] & = & \left(\begin{array}{c}\textbf{VarElim}\\ \circ\\ \textbf{Complete}\end{array}\right)\left[\begin{array}{l}p_1 :: pats_1, rhs_1\\ \quad\vdots\\ p_n :: pats_n, rhs_n\end{array}\right]\\
 & M_1 & = & \mathcal{P}\left(\begin{array}{l}\quad \mathcal{V}_1 @ stack,\\ \left[\begin{array}{l}\overline{p_{11}} @ pats_{11}, rhs_{11}\\ \quad\vdots\\ \overline{p_{1k_1}} @ pats_{1k_1}, rhs_{1k_1}\end{array}\right]\end{array}\right)\\
 & \vdots & & \\
 & M_n & = & \mathcal{P}\left(\begin{array}{l}\quad \mathcal{V}_n @ stack,\\ \left[\begin{array}{l}\overline{p_{n1}} @ pats_{n1}, rhs_{n1}\\ \quad\vdots\\ \overline{p_{nk_n}} @ pats_{nk_n}, rhs_{nk_n}\end{array}\right]\end{array}\right)\\
\mathtt{in} & & & \\
\multicolumn{4}{l}{\quad \mathsf{case\_ty}\ (\boldsymbol{\lambda}\mathcal{V}_1.\ M_1)\ldots(\boldsymbol{\lambda}\mathcal{V}_n.\ M_n)\ z}
\end{array}$$

End

$$\mathcal{P}([\,],[([\,],[rhs_1,\ldots,rhs_k])]) = rhs_1$$

Figure 3.4: Pattern matching algorithm

The termination and correctness of a similar algorithm is shown in [67], which deals with compilation of lazy pattern matching, a more complex problem that subsumes ours.

Example

We now exercise $\boldsymbol{\mathcal{P}}$ on our motivating example.

$$\boldsymbol{\mathcal{P}}\left(\begin{array}{l}{[z]},\\ \left[\begin{array}{ll}{[(x,\mathsf{0})]}, & \mathsf{True}\\ {[(\mathsf{0},y)]}, & \mathsf{False}\end{array}\right]\end{array}\right)$$

Since both rows are headed by pairs, the **Constructor** rule applies.

$$\mathsf{pair\_case}(\boldsymbol{\lambda} v_1\ v_2.\boldsymbol{\mathcal{P}}\left(\begin{array}{l}{[v_1,v_2]},\\ \left[\begin{array}{ll}{[x,\mathsf{0}]}, & \mathsf{True}\\ {[\mathsf{0},y]}, & \mathsf{False}\end{array}\right]\end{array}\right))\ z$$

The lead column is a mixture of variables and constructors, so a **VarElim** step is made.

$$\mathsf{pair\_case}(\boldsymbol{\lambda} v_1\ v_2.\boldsymbol{\mathcal{P}}\left(\begin{array}{l}{[v_1,v_2]},\\ \left[\begin{array}{ll}{[\mathsf{0},\mathsf{0}]}, & \mathsf{True}\\ {[\mathsf{Suc}\ v_3,\mathsf{0}]}, & \mathsf{True}\\ {[\mathsf{0},y]}, & \mathsf{False}\end{array}\right]\end{array}\right))\ z$$

The lead column is all constructor applications, so a **Constructor** step is made, splitting the translation into two cases.

$$\begin{array}{l}\mathsf{pair\_case}(\boldsymbol{\lambda} v_1\ v_2.\ \mathsf{num\_case}\\ \left(\dfrac{\boldsymbol{\mathcal{P}}\left(\begin{array}{l}{[v_2]},\\ \left[\begin{array}{ll}{[\mathsf{0}]}, & \mathsf{True}\\ {[y]}, & \mathsf{False}\end{array}\right]\end{array}\right)}{\left(\boldsymbol{\lambda} v_4.\ \boldsymbol{\mathcal{P}}\left(\begin{array}{l}{[v_4,v_2]},\\ \left[\begin{array}{ll}{[v_3,\mathsf{0}]}, & \mathsf{True}\end{array}\right]\end{array}\right)\right)}\right))\ v_1\ z\end{array}$$

The top case requires an application of **VarElim** and the bottom case invokes the **Variable** rule. In the latter application, the substitution $[v_3 \mapsto v_4]$ has no effect because the right hand side is a constant.

$$\begin{array}{l}\mathsf{pair\_case}(\boldsymbol{\lambda} v_1\ v_2.\ \mathsf{num\_case}\\ \left(\dfrac{\boldsymbol{\mathcal{P}}\left(\begin{array}{l}{[v_2]},\\ \left[\begin{array}{ll}{[\mathsf{0}]}, & \mathsf{True}\\ {[\mathsf{0}]}, & \mathsf{False}\\ {[\mathsf{Suc}\ v_5]}, & \mathsf{False}\end{array}\right]\end{array}\right)}{\left(\boldsymbol{\lambda} v_4.\ \boldsymbol{\mathcal{P}}\left(\begin{array}{l}{[v_2]},\\ \left[\begin{array}{ll}{[\mathsf{0}]}, & \mathsf{True}\end{array}\right]\end{array}\right)\right)}\right))\ v_1\ z\end{array}$$

The **Constructor** rule now applies above, and the **Complete** rule below.

$$\mathsf{pair\_case}(\boldsymbol{\lambda} v_1\ v_2.\ \mathsf{num\_case}$$
$$\left(\frac{\mathsf{num\_case}\left(\frac{\mathcal{P}\left(\begin{array}{l}{[\,]},\\ \left[\begin{array}{ll}{[\,]}, & \mathsf{True}\\ {[\,]}, & \mathsf{False}\end{array}\right]\end{array}\right)}{\boldsymbol{\lambda} v_7.\ \mathcal{P}\left(\begin{array}{l}{[v_7]},\\ \left[\begin{array}{ll}{[\,v_5]}, & \mathsf{False}\end{array}\right]\end{array}\right)}\right)\ v_2}{\left(\boldsymbol{\lambda} v_4.\ \mathcal{P}\left(\begin{array}{l}{[v_2]},\\ \left[\begin{array}{ll}{[0]}, & \mathsf{True}\\ {[\mathsf{Suc}\ v_6]}, & \mathsf{Arb}\end{array}\right]\end{array}\right)\right)}\right)\ )\ v_1\ z$$

Things finally become simpler by application of the **End** and **Variable** rules.

$$\mathsf{pair\_case}(\boldsymbol{\lambda} v_1\ v_2.\ \mathsf{num\_case}\ (\mathsf{num\_case}\ \mathsf{True}\ (\boldsymbol{\lambda} v_7.\ \mathsf{False})\ v_2)$$
$$\left(\boldsymbol{\lambda} v_4.\ \mathcal{P}\left(\begin{array}{l}{[v_2]},\\ \left[\begin{array}{ll}{[0]}, & \mathsf{True}\\ {[\mathsf{Suc}\ v_6]}, & \mathsf{Arb}\end{array}\right]\end{array}\right)\right)\ )\ v_1\ z$$

There is now a final application of the **Constructor** rule.

$$\mathsf{pair\_case}(\boldsymbol{\lambda} v_1\ v_2.\ \mathsf{num\_case}\ (\mathsf{num\_case}\ \mathsf{True}\ (\boldsymbol{\lambda} v_7.\ \mathsf{False})\ v_2)$$
$$\left(\boldsymbol{\lambda} v_4.\ \mathsf{num\_case}\left(\frac{\mathcal{P}\left(\begin{array}{l}{[\,]},\\ \left[\begin{array}{ll}{[\,]}, & \mathsf{True}\end{array}\right]\end{array}\right)}{\boldsymbol{\lambda} v_8.\ \mathcal{P}\left(\begin{array}{l}{[v_8]}\\ \left[\begin{array}{ll}{[v_6]}, & \mathsf{Arb}\end{array}\right]\end{array}\right)}\right)\ v_2\right)\ )\ v_1\ z$$

An application of the **Variable** rule and then a couple of applications of **End** complete the example. We finish by abstracting the initial element in the variable stack, and the function:

$$\begin{array}{l}\boldsymbol{\lambda} f\ z.\ \mathsf{pair\_case}(\boldsymbol{\lambda} v_1\ v_2.\ \mathsf{num\_case}\ (\mathsf{num\_case}\ \mathsf{True}\ (\boldsymbol{\lambda} v_7.\ \mathsf{False})\ v_2)\\ \qquad\qquad\qquad\qquad (\boldsymbol{\lambda} v_4.\ \mathsf{num\_case}\ \mathsf{True}\ (\boldsymbol{\lambda} v_8.\ \mathsf{Arb})\ v_2)\ v_1)\ z.\end{array}$$

□

This is equal to the result of calling algorithm $\boldsymbol{\mathcal{O}}$ on the following input:

$$\begin{array}{rcl} f(0,0) & \equiv & \mathsf{True}\\ f(\mathsf{Suc}\ x,0) & \equiv & \mathsf{True}\\ f(0,\mathsf{Suc}\ x) & \equiv & \mathsf{False}\\ f(\mathsf{Suc}\ x,\mathsf{Suc}\ y) & \equiv & \mathsf{Arb}.\end{array}$$

3.4 Customized induction theorems

It is a well-known consequence of the wellfounded induction theorem that, once a function has been shown to terminate by relation R, a customized induction theorem can be built for it. These are the 'induction schemes' of Boyer and Moore. In this section we show in detail how such induction theorems can be formally derived, using only the termination conditions obtained via the algorithm of Section 3.2.

3.4.1 Deriving induction

From a function definition

$$\begin{array}{rcl} f(pat_1) & \equiv & rhs_1[f(a_{11}), \ldots, f(a_{1k_1})] \\ & \vdots & \\ f(pat_n) & \equiv & rhs_n[f(a_{n1}), \ldots, f(a_{nk_n})] \end{array}$$

the steps of the definition algorithm of Section 3.1 produce the following termination conditions:

$$\begin{array}{lll} \mathsf{WF}(R), & & \\ \forall(\Gamma(a_{11}) \supset R\ a_{11}\ pat_1), & \ldots, & \forall(\Gamma(a_{1k_1}) \supset R\ a_{1k_1}\ pat_1), \\ & \vdots & \\ \forall(\Gamma(a_{n1}) \supset R\ a_{n1}\ pat_n), & \ldots, & \forall(\Gamma(a_{nk_n}) \supset R\ a_{nk_n}\ pat_n) \end{array}$$

(Note that, if the input patterns were incomplete or overlapping, that has been remedied by the operation of algorithm $\mathcal{P}$.) The termination conditions are used to generate the form of the desired induction theorem for f (displayed in Figure 3.5).

Subtle point revisited. In Section 3.2, a minimalist approach to quantification of termination conditions was argued for: the only variables universally quantified in a termination condition are those introduced on the right hand side of the definition. This allows recursion equations to be unrolled at a specific instance, provided the termination conditions can be proved at that instance. For the derivation of induction, a different approach is taken since each case in the target induction theorem should be fully quantified (excepting only the induction predicate P). The derivation of induction assumes the termination conditions at a crucial point; if they are not fully quantified, the derivation may fail. Thus the hypotheses of the induction theorem will be fully generalized versions of the hypotheses of the recursion equations. A fully quantified formula M will be displayed as $\boldsymbol{\forall}(M)$, in contrast to a minimally quantified formula $\forall(M)$.
□

$$\begin{array}{ll}
\left(\forall\left(\begin{array}{ll}(\forall(\Gamma(a_{11}) \supset P\ a_{11})) & \wedge \\ \vdots & \wedge \\ (\forall(\Gamma(a_{1k_1}) \supset P\ a_{1k_1})) & \end{array}\right) \supset P(pat_1)\right) & \wedge \\
 & \vdots \\
 & \wedge \\
\left(\forall\left(\begin{array}{ll}(\forall(\Gamma(a_{n1}) \supset P\ a_{n1})) & \wedge \\ \vdots & \wedge \\ (\forall(\Gamma(a_{nk_n}) \supset P\ a_{nk_n})) & \end{array}\right) \supset P(pat_n)\right) & \supset \forall v.\ P\ v.
\end{array}$$

Figure 3.5: Shape of target induction theorem

The general plan of the mechanized derivation of this theorem is to establish that the antecedent of the target induction theorem implies the antecedent of wellfounded induction (Theorem 6):

$$\begin{array}{ll}
\left(\forall\left(\begin{array}{l}(\forall(\Gamma(a_{11}) \supset P\ a_{11})) \\ \wedge \ldots \wedge \\ (\forall(\Gamma(a_{1k_1}) \supset P\ a_{1k_1}))\end{array}\right) \supset P(pat_1)\right) & \wedge \\
 & \vdots \\
 & \wedge \\
\left(\forall\left(\begin{array}{l}(\forall(\Gamma(a_{n1}) \supset P\ a_{n1})) \\ \wedge \ldots \wedge \\ (\forall(\Gamma(a_{nk_n}) \supset P\ a_{nk_n}))\end{array}\right) \supset P(pat_n)\right) & \supset \forall x.(\forall y.R\ y\ x \supset P\ y) \supset P\ x
\end{array}$$

Then by transitivity, we will have established the target induction theorem. In detail, the proof proceeds according to the following algorithm.

1. Assume the antecedent of the target theorem.

$$(1) \vdash \begin{array}{ll}
\left(\forall\left(\begin{array}{ll}(\forall(\Gamma(a_{11}) \supset P\ a_{11})) & \wedge \\ \vdots & \wedge \\ (\forall(\Gamma(a_{1k_1}) \supset P\ a_{1k_1})) & \end{array}\right) \supset P(pat_1)\right) & \wedge \\
 & \vdots \\
 & \wedge \\
\left(\forall\left(\begin{array}{ll}(\forall(\Gamma(a_{n1}) \supset P\ a_{n1})) & \wedge \\ \vdots & \wedge \\ (\forall(\Gamma(a_{nk_n}) \supset P\ a_{nk_n})) & \end{array}\right) \supset P(pat_n)\right) &
\end{array}$$

2. Consider cases on $pat_1, \ldots, pat_n$.

   (a) For case pat_i, assume the instantiated antecedent of Theorem 6:

   $$\forall y.\ R\ y\ pat_i \supset P\ y \vdash \forall y.\ R\ y\ pat_i \supset P\ y.$$

   (b) Specialize (a) to each recursive argument in the clause, obtaining

   $$\begin{array}{ccl} (a) & \vdash & R\ a_{i1}\ pat_i \supset P\ a_{i1} \\ & \vdots & \\ (a) & \vdash & R\ a_{ik_i}\ pat_i \supset P\ a_{ik_i}. \end{array}$$

   (c) Prove each antecedent of the theorems from (b): first, assume the fully quantified termination condition and then specialize it fully.

   $$\begin{array}{rcl} \boldsymbol{\forall}(\Gamma(a_{i1}) \supset R\ a_{i1}pat_i) & \vdash & \Gamma(a_{i1}) \supset R\ a_{i1}pat_i \\ & \vdots & \\ \boldsymbol{\forall}(\Gamma(a_{ik_i}) \supset R\ a_{ik_i}pat_i) & \vdash & \Gamma(a_{ik_i}) \supset R\ a_{ik_i}pat_i. \end{array}$$

   Now use transitivity of $\supset$ (or just *modus ponens*, when the context is empty), to eliminate $R\ a_{i1}\ pat_i, \ldots, R\ a_{ik_i}\ pat_i$ to get

   $$\begin{array}{rcl} (a), \boldsymbol{\forall}(\Gamma(a_{i1}) \supset R\ a_{i1}pat_i) & \vdash & \Gamma(a_{i1}) \supset P\ a_{i1} \\ & \vdots & \\ (a), \boldsymbol{\forall}(\Gamma(a_{ik_i}) \supset R\ a_{ik_i}pat_i) & \vdash & \Gamma(a_{ik_i}) \supset P\ a_{ik_i}. \end{array}$$

   (d) Now generalize only the local variables of each theorem in (c). Quantification is over the free variables in the context of the recursive call, but not including the variables in the pattern, *i.e.*, $FV(\Gamma(a_{ij}) \supset P\ a_{ij}) - FV(pat_i)$. This allows for the proper quantification of variables local to the right-hand side of a clause in a function definition, *e.g.*, those introduced by a let binding.

   $$\begin{array}{rcl} (a), \boldsymbol{\forall}(\Gamma(a_{i1}) \supset R\ a_{i1}pat_i) & \vdash & \forall(\Gamma(a_{i1}) \supset P\ a_{i1}) \\ & \vdots & \\ (a), \boldsymbol{\forall}(\Gamma(a_{ik_i}) \supset R\ a_{ik_i}pat_i) & \vdash & \forall(\Gamma(a_{ik_i}) \supset P\ a_{ik_i}). \end{array}$$

   (e) By *modus ponens* with the ith conjunct of (1) and the conjunction of the theorems from (d) we have

   $$\left[\begin{array}{rccl} (a), (1), & \boldsymbol{\forall}(\Gamma(a_{i1}) & \supset & R\ a_{i1}pat_i), \\ & & \vdots & \\ & \boldsymbol{\forall}(\Gamma(a_{ik_i}) & \supset & R\ a_{ik_i}pat_i) \end{array}\right] \vdash P\ pat_i$$

and by discharging (a), we obtain

$$\left[\begin{array}{lcl} (1),\ \boldsymbol{\forall}(\Gamma(a_{i1}) & \supset & R\ a_{i1} pat_i), \\ & \vdots & \\ \qquad \boldsymbol{\forall}(\Gamma(a_{ik_i}) & \supset & R\ a_{ik_i} pat_i) \end{array}\right] \vdash (\forall y.\ R\ y\ pat_i \supset P\ y) \supset P\ pat_i$$

(f) Replace pat_i by x. Note that the strong quantification of the hypotheses means that the replacement only affects the conclusion.

$$\left[\begin{array}{lcl} x = pat_i,\ (1), & & \\ \quad \boldsymbol{\forall}(\Gamma(a_{i1}) & \supset & R\ a_{i1} pat_i), \\ & \vdots & \\ \quad \boldsymbol{\forall}(\Gamma(a_{ik_i}) & \supset & R\ a_{ik_i} pat_i) \end{array}\right] \vdash (\forall y.\ R\ y\ x \supset P\ y) \supset P\ x$$

3. Steps a to f have now been done for each case, so the following n theorems have been proved:

$$\left[\begin{array}{lcl} x = pat_1,\ (1), & & \\ \quad \boldsymbol{\forall}(\Gamma(a_{11}) & \supset & R\ a_{11} pat_1), \\ & \vdots & \\ \quad \boldsymbol{\forall}(\Gamma(a_{1k_1}) & \supset & R\ a_{1k_1} pat_1) \end{array}\right] \vdash (\forall y.\ R\ y\ x \supset P\ y) \supset P\ x$$

$$\vdots$$

$$\left[\begin{array}{lcl} x = pat_n,\ (1), & & \\ \quad \boldsymbol{\forall}(\Gamma(a_{n1}) & \supset & R\ a_{n1} pat_n), \\ & \vdots & \\ \quad \boldsymbol{\forall}(\Gamma(a_{nk_n}) & \supset & R\ a_{nk_n} pat_n) \end{array}\right] \vdash (\forall y.\ R\ y\ x \supset P\ y) \supset P\ x$$

4. We now need a disjunction theorem

$$\vdash \forall x.\ (\exists \overline{y_1}.\ x = pat_1) \vee \ldots \vee (\exists \overline{y_n}.\ x = pat_n)$$

where the free variables of pat_i in disjunct i comprise the vector y_i. We will consider the production of this theorem in Section 3.4.2.

5. By applying a disjoint cases rule scheme to (3) and (4) and then generalizing x, we obtain

$$\left[\begin{array}{lcl} (1),\ \boldsymbol{\forall}(\Gamma(a_{11}) & \supset & R\ a_{11} pat_1), \\ & \vdots & \\ \qquad \boldsymbol{\forall}(\Gamma(a_{1k_1}) & \supset & R\ a_{1k_1} pat_1), \\ \qquad \vdots & & \\ \qquad \boldsymbol{\forall}(\Gamma(a_{n1}) & \supset & R\ a_{n1} pat_n), \\ & \vdots & \\ \qquad \boldsymbol{\forall}(\Gamma(a_{nk_n}) & \supset & R\ a_{nk_n} pat_n) \end{array}\right] \vdash \forall x.\ (\forall y.\ R\ y\ x \supset P\ y) \supset P\ x.$$

If the hypotheses had not been strongly quantified, this step would fail.

6. Now by *modus ponens* with the wellfounded induction theorem and (5), we have derived $\mathsf{WF}(R), \ldots, (1) \vdash \forall x.\ P\ x$, as planned. Discharge the assumption (1) and then generalize to get the target theorem:

$$
\boxed{
\begin{array}{l}
\left[\begin{array}{ccc}
 & \mathsf{WF}(R), & \\
\forall(\Gamma(a_{11}) \supset R\ a_{11} pat_1), & \ldots, & \forall(\Gamma(a_{1k_1}) \supset R\ a_{1k_1} pat_1), \\
 & \vdots & \\
\forall(\Gamma(a_{n1}) \supset R\ a_{n1} pat_n), & \ldots, & \forall(\Gamma(a_{nk_n}) \supset R\ a_{nk_n} pat_n)
\end{array}\right] \\
\vdash \\
\left(\forall \left(\begin{array}{ll}
(\forall(\Gamma(a_{11}) \supset P\ a_{11})) & \wedge \\
\vdots & \wedge \\
(\forall(\Gamma(a_{1k_1}) \supset P\ a_{1k_1})) &
\end{array}\right) \supset P(pat_1)\right) \quad \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \\
\left(\forall \left(\begin{array}{ll}
(\forall(\Gamma(a_{n1}) \supset P\ a_{n1})) & \wedge \\
\vdots & \wedge \\
(\forall(\Gamma(a_{nk_n}) \supset P\ a_{nk_n})) &
\end{array}\right) \supset P(pat_n)\right) \quad \supset \forall v.\ P\ v.
\end{array}
}
$$

□

Remark. Since the derivation of induction schemes is driven solely by the form of the termination conditions, some definitions may have overly elaborate schemes. For example, the definition of gcd

$$
\begin{aligned}
\mathsf{gcd}(0, y) &\equiv y \\
\mathsf{gcd}(\mathsf{Suc}\ x, 0) &\equiv \mathsf{Suc}\ x \\
\mathsf{gcd}(\mathsf{Suc}\ x, \mathsf{Suc}\ y) &\equiv \mathtt{if}\ y \leq x\ \mathtt{then}\ \mathsf{gcd}(x - y,\ \mathsf{Suc}\ y)\ \mathtt{else}\ \mathsf{gcd}(\mathsf{Suc}\ x,\ y - x),
\end{aligned}
$$

yields the following induction theorem:

$$
\begin{array}{ll}
\forall P. & (\forall y.\ P\ (0, y)) \wedge \\
 & (\forall x.\ P\ (\mathsf{Suc}\ x, 0)) \wedge \\
 & (\forall x\ y.\ (\neg(y \leq x) \supset P\ (\mathsf{Suc}\ x, y - x)) \wedge \\
 & \qquad\qquad (y \leq x\ \supset P\ (x - y, \mathsf{Suc}\ y)) \supset P(\mathsf{Suc}\ x, \mathsf{Suc}\ y)) \\
 & \supset \forall v\ v_1.\ P\ (v, v_1).
\end{array}
$$

However, a simpler scheme that omits the guards on recursive calls is also derivable:

$$\begin{array}{ll}\forall P. & (\forall y.\ P(0, y)) \land \\ & (\forall x.\ P(\mathsf{Suc}\ x, 0)) \land \\ & (\forall x\ y.\ P(\mathsf{Suc}\ x, y - x) \land P(x - y, \mathsf{Suc}\ y) \supset P(\mathsf{Suc}\ x, \mathsf{Suc}\ y)) \\ & \supset \forall v\ v_1.\ P\ (v, v_1).\end{array}$$

In general, if the termination conditions for a recursive call can be proven without reference to the context of the call, the induction scheme need not guard the recursive call. We do not take such considerations into account in the derivation of the induction scheme, because we will in general want to derive induction schemes without having any knowledge about the termination proof of the function. In Section 3.5 we explain how this enables the derivation of induction schemes for recursive functions that have not had a termination relation supplied.

3.4.2 Proving completeness of patterns

The reader may wonder why pattern completeness needs to be proven for the induction theorem. After all, the termination conditions used as input to the derivation of the induction theorem ultimately come from the functional that has been built by algorithm $\mathcal{P}$. Recall that $\mathcal{P}$ automatically completes and removes overlaps from any pattern set it is given. Thus we already know—if we believe the metalanguage calculation—that the induction theorem must be pattern complete. However, since we use a formal logic that lacks the ability to import the results of metalanguage calculation as theorems, *e.g.*, a computational *reflection* mechanism [4, 49], this calculation must also be carried out as an object language deduction; to, in effect, check the work of algorithm $\mathcal{P}$.

The following algorithm $\mathcal{Q}$, is a slightly adapted version of algorithm $\mathcal{O}$. The essence of its operation is that it recursively rebuilds the structure of the given patterns in a top-down manner. When the recursion bottoms out, the variables in the patterns are existentially quantified. As the recursion unwinds, the exhaustion theorems for datatypes are used to combine the theorems coming from subcases (this is the counterpart of the step of generating a case expression in $\mathcal{O}$). The end result is a single theorem expressing the completeness of the pattern set.

As with $\mathcal{O}$, $\mathcal{Q}$ takes two arguments: a variable stack and a list of *rows*. A row is now a triple: a list of patterns, a theorem, and a list of variable bindings. The theorem component is where the structure of the patterns gets built. The variable bindings are used to associate the variable names in the original patterns with freshly generated variables. Assume we are given the complete and non-overlapping patterns $pat_1, \ldots, pat_n$. The algorithm starts by building the rows and initializing the variable stack to a variable z which does not occur free in the given patterns. Then the $\mathcal{Q}$ function is called: the resulting theorem is generalized with respect to x.

$$\vdash \forall x.\ \mathcal{Q}\left([z], \begin{bmatrix} ([pat_1],\ (x = z \vdash x = z), []), \\ \vdots \\ ([pat_n],\ (x = z \vdash x = z), []) \end{bmatrix} \right)$$

As with $\mathcal{O}$, there are only three cases to consider when examining the lead column: (1) the column is all variables, (2) the column is all constructors, and (3) the column is empty.

The **Variable** rule covers the case in which the current pattern being examined (in all rows) is a variable. The notation $\langle u, v\rangle$ denotes a variable binding, which is carried through the computation until the **End** case is encountered. Explicitly carrying the substitution along avoids technical problems stemming from name clashes; otherwise, the substitution $[z \mapsto v_i]$ could be performed in the theorem component of each row i.

Variable.

$$\begin{array}{rlll} & \mathcal{Q}(z :: \mathit{stack}, & [(v_1 :: \mathit{pats}_1, \mathit{th}, \theta) & \ldots,\ (v_n :: \mathit{pats}_n, \mathit{th}, \theta)]) \\ = & \mathcal{Q}(\mathit{stack}, & [(\mathit{pats}_1, \mathit{th}, \langle z, v_1\rangle :: \theta), & \ldots,\ (\mathit{pats}_n, \mathit{th}, \langle z, v_n\rangle :: \theta)]) \end{array}$$

In the **End** rule, the pattern has been completely rebuilt and the variable stack is empty. The theorem is now existentially quantified with all the variables in the binding list (in effect, each variable is getting renamed to its original in the pattern). Notice that the existential quantification detaches the constraints held in the hypotheses of the theorem from their occurrence in the conclusion.

End

$$\begin{array}{l} \mathcal{Q}([\,],\ [([\,],\ (\Gamma \vdash x = M[x_1, \ldots, x_j]),\ [\langle x_1, y_1\rangle, \ldots, \langle x_j, y_j\rangle])]) \\ = \ \Gamma \vdash \exists y_1 \ldots y_j.\ x = M[y_1, \ldots, y_j]. \end{array}$$

In the **Constructor** rule, the current sub-pattern being examined in all patterns is a constructor for a type τ. As in $\mathcal{O}$, the problem is partitioned into n subproblems of size $k_1 \ldots k_n$. We begin in the following situation:

Constructor (1)

$$\boldsymbol{\mathcal{Q}}\left(\begin{array}{l} z :: stack, \\ \left[\begin{array}{ll} (\mathsf{C}_1\overline{p}_{11} :: pats_{11}, & \Gamma \vdash x = N, \theta_{11}) \\ \quad\dots & \\ (\mathsf{C}_1\overline{p}_{1k_1} :: pats_{1k_1}, & \Gamma \vdash x = N, \theta_{1k_1}), \\ & \vdots \\ (\mathsf{C}_n\overline{p}_{n1} :: pats_{n1}, & \Gamma \vdash x = N, \theta_{n1}) \\ \quad\dots & \\ (\mathsf{C}_n\overline{p}_{nk_n} :: pats_{nk_n}, & \Gamma \vdash x = N, \theta_{nk_n}) \end{array}\right] \end{array}\right)$$

Note that the theorem $\Gamma \vdash x = N$ is the same in all rows, since it is getting 'pushed down' towards the leaves, and being augmented in the process. A case split on the ways to construct τ is made, and $\boldsymbol{\mathcal{Q}}$ is called on each case. In subproblem i, supposing the constructor C_i has type $\tau_1 \to \dots \to \tau_j \to \tau$, j new variables $v_1 : \tau_1, \dots, v_j : \tau_j$ are pushed onto the stack. This vector of variables is denoted $\overline{v_i}$.

Constructor (2)

$$\begin{array}{l} \boldsymbol{\mathcal{Q}}\left(\begin{array}{l} \overline{v_1}@stack, \\ \left[\begin{array}{lll} (\overline{p}_{11}@pats_{11}, & \Gamma, z = C_1\overline{v_1} \vdash x = [z \mapsto C_1\overline{v_1}]N, & \theta_{11}), \\ \quad\vdots & & \\ (\overline{p}_{1k_1}@pats_{1k_1}, & \Gamma, z = C_1\overline{v_1} \vdash x = [z \mapsto C_1\overline{v_1}]N, & \theta_{1k_1}) \end{array}\right] \end{array}\right) \\ \vdots \\ \boldsymbol{\mathcal{Q}}\left(\begin{array}{l} \overline{v_n}@stack, \\ \left[\begin{array}{lll} (\overline{p}_{n1}@pats_{n1}, & \Gamma, z = C_n\overline{v_n} \vdash x = [z \mapsto C_n\overline{v_n}]N, & \theta_{n1}) \\ \quad\vdots & & \\ (\overline{p}_{nk_n}@pats_{nk_n}, & \Gamma, z = C_n\overline{v_n} \vdash x = [z \mapsto C_n\overline{v_n}]N, & \theta_{nk_n}) \end{array}\right] \end{array}\right) \end{array}$$

The recursive calls delivers the following theorems.

Constructor (3)

$$\begin{array}{rcl} \Gamma, z = C_1\overline{y} & \vdash & (\exists\overline{y}.\ x = M_{11}[\overline{y}]) \vee \dots \vee (\exists\overline{y}.\ x = M_{1k_1}[\overline{y}]) \\ & \vdots & \\ \Gamma, z = C_n\overline{y} & \vdash & (\exists\overline{y}.\ x = M_{n1}[\overline{y}]) \vee \dots \vee (\exists\overline{y}.\ x = M_{nk_n}[\overline{y}]) \end{array}$$

The cases can be existentially quantified on the left and disjoined such that all conclusions are equal:

Constructor (4)

$$\begin{array}{rcllll}
\Gamma, \exists\overline{y}.\ z = C_1\overline{y} & \vdash & (\exists\overline{y}.\ x = M_{11}[\overline{y}]) \lor & \ldots & \lor\ (\exists\overline{y}.\ x = M_{1k_1}[\overline{y}]) & \lor \\
 & & & \vdots & & \lor \\
 & & (\exists\overline{y}.\ x = M_{n1}[\overline{y}]) \lor & \ldots & \lor\ (\exists\overline{y}.\ x = M_{nk_n}[\overline{y}]) & \\
 & \vdots & & & & \\
\Gamma, \exists\overline{y}.\ z = C_n\overline{y} & \vdash & (\exists\overline{y}.\ x = M_{11}[\overline{y}]) \lor & \ldots & \lor\ (\exists\overline{y}.\ x = M_{1k_1}[\overline{y}]) & \lor \\
 & & & \vdots & & \lor \\
 & & (\exists\overline{y}.\ x = M_{n1}[\overline{y}]) \lor & \ldots & \lor\ (\exists\overline{y}.\ x = M_{nk_n}[\overline{y}]) &
\end{array}$$

Finally, a disjoint cases rule scheme is invoked with the exhaustion theorem for τ:

$$\forall x : \tau.\ (\exists\overline{y}.x = \mathsf{C}_1\overline{y}) \lor \ldots \lor (\exists\overline{y}.x = \mathsf{C}_n\overline{y}).$$

This delivers the completeness of the original pattern set, with respect to the case analysis available at type τ.

Constructor (5)

$$\begin{array}{rllll}
\Gamma \vdash & (\exists\overline{y}.\ x = M_{11}[\overline{y}]) \lor & \ldots & \lor\ (\exists\overline{y}.\ x = M_{1k_1}[\overline{y}]) & \lor \\
 & & \vdots & & \lor \\
 & (\exists\overline{y}.\ x = M_{n1}[\overline{y}]) \lor & \ldots & \lor\ (\exists\overline{y}.\ x = M_{nk_n}[\overline{y}]) &
\end{array}$$

3.4.3 Remarks

The significance of our induction theorems is that they are algorithmically derived from the presentation of the function. In extracting the termination conditions, and then producing the induction theorem, a valuable piece of static analysis has been performed: the customized induction theorem tells, for any property, what cases must hold and what assumptions can be made in each case in trying to prove the property. Furthermore, the induction theorems are relatively general and thus perhaps might be useful as a means of classifying functions. The classification of functions would give a means of saying which functions were 'similar' to others, based on the shape of the induction scheme computed for the function.

Another aspect that may be of worthy of further investigation is the relation between implicit and explicit proofs, as embodied by the pattern matching algorithms $\mathcal{O}$ and $\mathcal{Q}$, in which the control structure is exactly the same; only the data manipulations differ. One question to ask is the following: is there a difference in the asymptotic complexity of the two algorithms? If so, this is a good example of a challenge that TFL in particular, and LCF style theorem provers in general, must overcome in order to handle large examples.

3.5 Definitions without termination relations

The principal obstacle in using our principle of definition for wellfounded recursion is the requirement that a wellfounded relation be given at definition time: if a correct one cannot be found, that means that the definition cannot be made, and that later definitions are jeopardized. Thus the automation of the *act* of definition is hampered. In this section, we show that there is a way to defer the obligation to give a correct termination relation; namely, if the termination relation R is left variable in the proof that captures the termination condition at a recursive call (in Figure 3.2), termination conditions can be extracted *before* making the definition. We must merely ensure, in step B of the proof, that R is not quantified. This allows all the termination conditions to be gathered and then a wellfounded relation satisfying them can be *chosen* before making the definition. In other words, extraction and definition commute. This allows one to omit termination relations for a large class of definitions. The computed termination conditions become constraints on the resulting recursion equations and the induction theorem.

The utility of this style of definition is that the task of proving termination can be dealt with in isolation from the rest of the formalization. Separately, the termination problems can be tackled by humans, or powerful automated systems specializing in termination, of the sort discussed in [38]. It is quite convenient to define functions in the manner of this section, an opinion that has been registered in a more general context by Girard [41, page 44]:

> The best thing is to work on recursive functions as if they were partial, and eventually remark afterward that the function is total.

However, a strong note of caution must be added: although a single function is described the moment the definition is made, one really wants to be able to use the recursion equations in an unconstrained manner. This avoids nasty surprises, *e.g.*, coming to the end of a long formal development only to find that a fundamental definition is not total. A verification cannot be considered finished if the final theorem has remaining termination conditions!

3.5.1 Relationless definition algorithm

The steps in a relationless definition will be quite similar to those of the algorithm from Section 3.1.

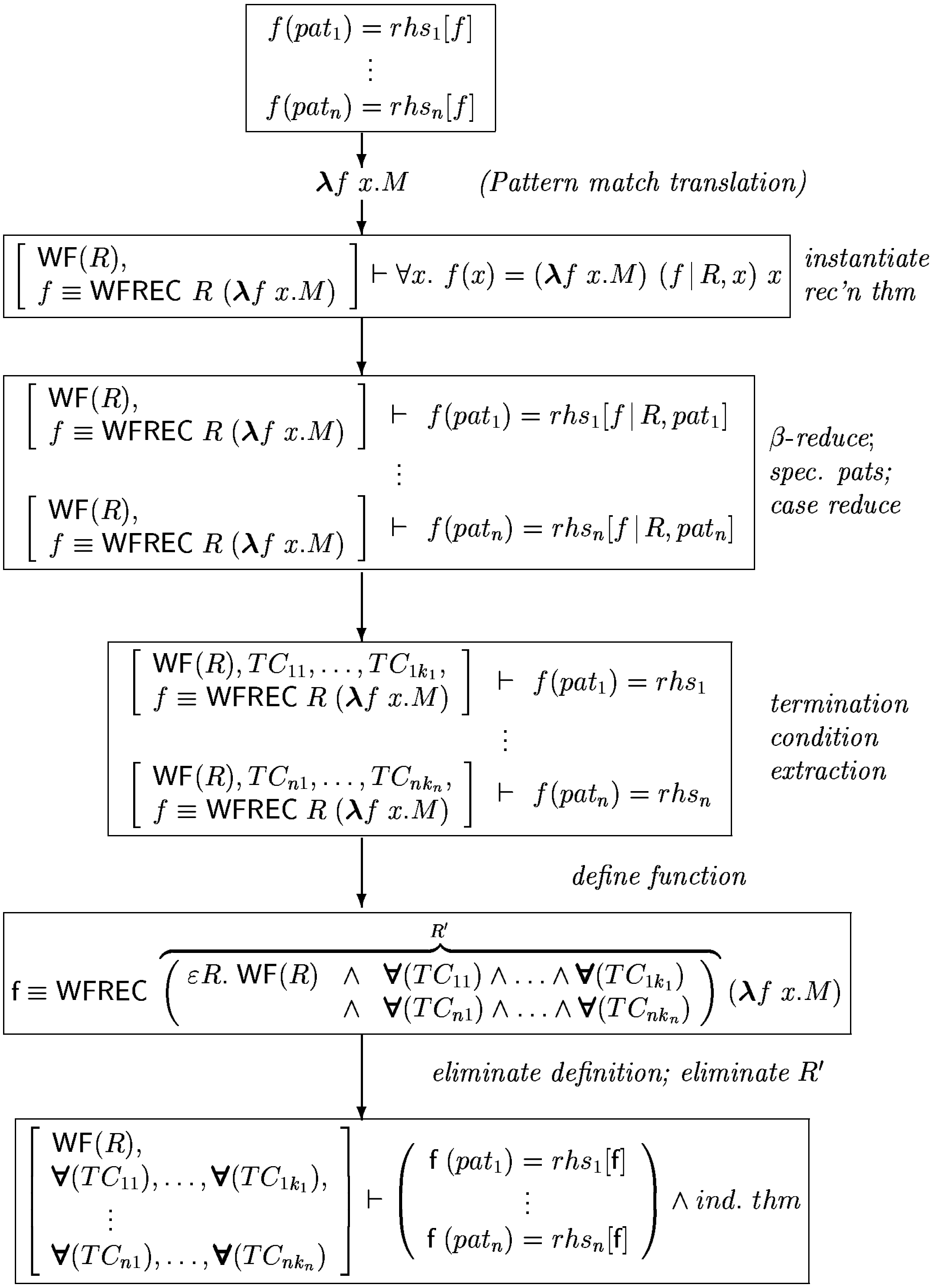

$f(pat_1) = rhs_1[f]$
$\vdots$
$f(pat_n) = rhs_n[f]$
$\boldsymbol{\lambda} f\ x.M$
(Pattern match translation)
$\left[\begin{array}{l}\mathsf{WF}(R),\\ f \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M)\end{array}\right] \vdash \forall x.\ f(x) = (\boldsymbol{\lambda} f\ x.M)\ (f \mid R, x)\ x$
instantiate rec'n thm
$\left[\begin{array}{l}\mathsf{WF}(R),\\ f \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M)\end{array}\right] \vdash\ f(pat_1) = rhs_1[f \mid R, pat_1]$
$\vdots$
$\left[\begin{array}{l}\mathsf{WF}(R),\\ f \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M)\end{array}\right] \vdash\ f(pat_n) = rhs_n[f \mid R, pat_n]$
β-reduce; spec. pats; case reduce
$\left[\begin{array}{l}\mathsf{WF}(R), TC_{11}, \ldots, TC_{1k_1},\\ f \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M)\end{array}\right] \vdash\ f(pat_1) = rhs_1$
$\vdots$
$\left[\begin{array}{l}\mathsf{WF}(R), TC_{n1}, \ldots, TC_{nk_n},\\ f \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M)\end{array}\right] \vdash\ f(pat_n) = rhs_n$
termination condition extraction
define function
$\mathsf{f} \equiv \mathsf{WFREC}\ \overbrace{\left(\begin{array}{ll}\varepsilon R.\ \mathsf{WF}(R) & \wedge\ \forall(TC_{11}) \wedge \ldots \wedge \forall(TC_{1k_1})\\ & \wedge\ \forall(TC_{n1}) \wedge \ldots \wedge \forall(TC_{nk_n})\end{array}\right)}^{R'}\ (\boldsymbol{\lambda} f\ x.M)$
eliminate definition; eliminate R'
$\left[\begin{array}{l}\mathsf{WF}(R),\\ \forall(TC_{11}), \ldots, \forall(TC_{1k_1}),\\ \quad\vdots\\ \forall(TC_{n1}), \ldots, \forall(TC_{nk_n})\end{array}\right] \vdash \left(\begin{array}{c}\mathsf{f}\ (pat_1) = rhs_1[\mathsf{f}]\\ \vdots\\ \mathsf{f}\ (pat_n) = rhs_n[\mathsf{f}]\end{array}\right) \wedge\ ind.\ thm$

Figure 3.6: Algorithm for relationless definition.

As before, the functional $\boldsymbol{\lambda} f\ x.M$ is calculated from the input recursion equations via the pattern-matching translation. Then the recursion theorem is instantiated with the functional. The wellfounded relation R is left as a variable. Now extraction is invoked on the instantiated recursion theorem. Extraction leaves, as before,

$$\mathsf{WF}(R), \forall(\Gamma(z_1) \supset R\ z_1 z), \ldots, \forall(\Gamma(z_n) \supset R\ z_n z) \vdash f(z) = M,$$

with the difference that both R and f are free variables. We fully quantify the other variables in each hypothesis. We write these termination conditions as $\mathsf{WF}(R), \boldsymbol{\forall}(TC_1(R)), \ldots, \boldsymbol{\forall}(TC_n(R))$, highlighting the fact that they have a single free variable R. Now we may define f by choosing a wellfounded relation meeting the termination conditions:

$$\mathsf{f} \equiv \mathsf{WFREC}\ (\varepsilon R.\ \mathsf{WF}(R) \land \boldsymbol{\forall}(TC_1(R)) \land \ldots \land \boldsymbol{\forall}(TC_n(R)))\ M.$$

(For the rest of this discussion, we take R' to abbreviate $(\varepsilon R.\mathsf{WF}(R) \land \boldsymbol{\forall}(TC_1(R)) \land \ldots \land \boldsymbol{\forall}(TC_n(R)))$.) Having made the definition, we can now instantiate the variable f by the constant f and eliminate the definition from the recursion theorem, giving the derived definition

$$\mathsf{WF}(R'), \boldsymbol{\forall}(TC_1(R')), \ldots, \boldsymbol{\forall}(TC_n(R')) \vdash \mathsf{f}\ z = M.$$

Now the termination conditions are assumed and conjoined, giving

$$\mathsf{WF}(R), \boldsymbol{\forall}(TC_1(R)), \ldots, \boldsymbol{\forall}(TC_n(R)) \vdash \mathsf{WF}(R) \land \boldsymbol{\forall}(TC_1(R)) \land \ldots \land \boldsymbol{\forall}(TC_n(R)),$$

to which we apply the Select Axiom $(\forall P\ x.\ P\ x \supset P(\varepsilon P))$, after which we can eliminate each R'-hypothesis from the derived definition. This leaves us with

$$\mathsf{WF}(R), \boldsymbol{\forall}(TC_1(R)), \ldots, \boldsymbol{\forall}(TC_n(R)) \vdash \mathsf{f}\ z = M,$$

a derived definition in which the computed termination conditions and the wellfoundedness requirement have been separated from the recursion equations and can be eliminated at the user's convenience. In order that the termination conditions be inert during inference, *i.e.*, fully closed, they may be existentially quantified:

$$(\exists R.\ \mathsf{WF}(R) \land \boldsymbol{\forall}(TC_1(R)) \land \ldots \land \boldsymbol{\forall}(TC_n(R))) \vdash \mathsf{f}\ z = M.$$

However, a perhaps cleaner alternative is to make a new definition

$$\mathsf{fTerminates} \equiv \exists R.\mathsf{WF}(R) \land \boldsymbol{\forall}(TC_1(R)) \land \ldots \land \boldsymbol{\forall}(TC_n(R))$$

and use that to return the theorem $\mathsf{fTerminates} \vdash \mathsf{f}\ z = M$. Such a naive attempt will fail when the termination conditions are polymorphic: a type variable in the right-hand side of the definition will not occur in the left-hand side of the definition (which has type bool). Thus, the best that can be done in the current HOL logic[1] is to make a parameterized definition:

$$\mathsf{fTerminates}\ R \equiv \mathsf{WF}(R) \wedge \forall(TC_1(R)) \wedge \ldots \wedge \forall(TC_n(R)).$$

Failure cases

The technique of this section fails in cases where the context of a call to f contains occurrences of f, because the termination conditions will have occurrences of f; the HOL definition principle will not allow the *definiens* to occur in the *definiendum*. Such cases occur when a nested function is being defined; a solution to the nestedness problem is given in Chapter 4. However, the technique also fails in some non-nested functions. For example, imagine that the task at hand is to define an evaluation function over a datatype (exp) containing a constructor for conditional expressions:

$$\mathsf{IF} : \mathsf{exp} \rightarrow \mathsf{exp} \rightarrow \mathsf{exp} \rightarrow \mathsf{exp}.$$

Ignoring the other clauses in the definition (and supposing for the sake of exposition that the evaluation function doesn't carry an environment with it), imagine that evaluation of a conditional in the type exp is interpreted by the pre-existing conditional in the logic:

$$\mathit{Eval}\ (\mathsf{IF}\ b\ e_1\ e_2) \equiv \mathtt{if}\ \mathit{Eval}\ b\ \mathtt{then}\ \mathit{Eval}\ e_1\ \mathtt{else}\ \mathit{Eval}\ e_2.$$

The three termination conditions extracted for this clause are

$$\begin{aligned} R\, b\ (\mathsf{IF}\ b\, e_1\ e_2), \\ \mathit{Eval}\ b \supset R\, e_1\ (\mathsf{IF}\ b\, e_1\ e_2), \\ \neg(\mathit{Eval}\ b) \supset R\, e_2\ (\mathsf{IF}\ b\, e_1\ e_2). \end{aligned}$$

The occurrences of *Eval* in the last two conditions cause the technique outlined above to fail. In such cases, the techniques of Chapter 4 can be used.

Remark. The derivation of the induction theorem for a relationless definition is exactly that of Section 3.4. The only extra work is that care must be taken that the termination relation R—a variable—is never quantified.

Remark. The algorithm of this section depends essentially on the indefinite nature of Hilbert's ε operator: *any* wellfounded relation satisfying the termination conditions must be acceptable, not a particular one, as would be required by a definite descriptor *e.g.*, as in the $\mathcal{Q}_0$ logic of Andrews [6]

[1] Melham's type quantifier proposal [73] could allow type variables to be quantified on the right hand side of the definition.

3.6 Schematic definitions

We can liberalize definitions still further by exploiting the form of the wellfounded recursion theorem. In making a definition

$$\mathsf{f} \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M)$$

the arguments to the function have thus far been bound solely in the second argument to the functional $\boldsymbol{\lambda} f\ x.M$, *i.e.*, in x. However, one can fruitfully distinguish between *arguments* to the function, and *parameters*. A parameter is defined to be a variable free in $\boldsymbol{\lambda} f\ x.M$. Attempting to directly invoke the principle of definition with a functional having such free variables will of course fail; however, one can parameterize the definition by the free variables $X_1, \ldots, X_j$ of the functional to obtain a valid definition:

$$\mathsf{f} \equiv \boldsymbol{\lambda} X_1 \ldots X_j.\ \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M).$$

This amounts to defining a recursive *schema*. Encouragingly, the commutation of definition and extraction from the previous section still holds, with minor modification. Likewise, the proof of induction for such definitions is only a trivial generalization of that given in Section 3.4.

Example

Consider the following description of the 'while' construct familiar from imperative programming:

$$\mathsf{While}\ s = \texttt{if}\ B\ s\ \texttt{then}\ \mathsf{While}\ (C\ s)\ \texttt{else}\ s.$$

Notice that the variables B and C occur only on the right hand side. Applying the pattern-matching translation gives:

$$\boldsymbol{\lambda} \mathit{While}\ s.\ \texttt{if}\ B\ s\ \texttt{then}\ \mathit{While}\ (C\ s)\ \texttt{else}\ s.$$

Instantiating the recursion theorem with this, and then performing the simplification steps and termination condition extraction leaves

$$\begin{array}{l} \left[\begin{array}{l} \mathsf{WF}\ R,\ \forall s.\ B\ s \supset R\ (C\ s)\ s, \\ f \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} \mathit{While}\ s.\ \texttt{if}\ B\ s\ \texttt{then}\ \mathit{While}\ (C\ s)\ \texttt{else}\ s) \end{array}\right] \\ \qquad \vdash \\ f\ s = \texttt{if}\ B\ s\ \texttt{then}\ f\ (C\ s)\ \texttt{else}\ s. \end{array}$$

Now the algorithm defines

$$\begin{array}{ll} \mathsf{While} \equiv \boldsymbol{\lambda} B\ C.\ \mathsf{WFREC} & (\varepsilon R.\ \mathsf{WF}\ R \land \forall s.\ B\ s \supset R\ (C\ s)\ s) \\ & (\boldsymbol{\lambda} \mathit{While}\ s.\ \texttt{if}\ B\ s\ \texttt{then}\ \mathit{While}\ (C\ s)\ \texttt{else}\ s). \end{array}$$

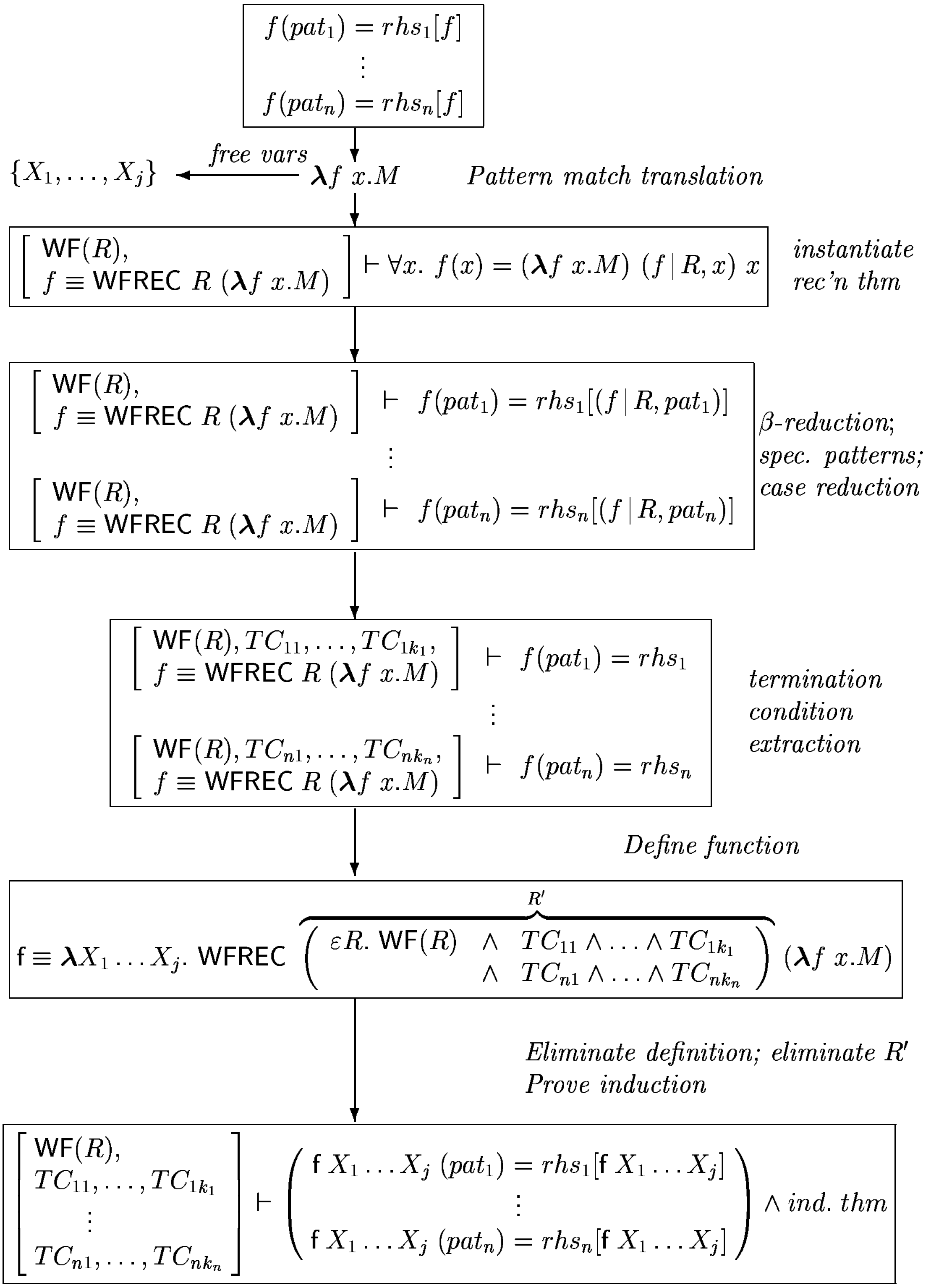

f(pat_1) = rhs_1[f] ... f(pat_n) = rhs_n[f]
free vars
{X_1, ..., X_j}
λf x.M
Pattern match translation
[WF(R), f ≡ WFREC R (λf x.M)] ⊢ ∀x. f(x) = (λf x.M) (f | R, x) x
instantiate rec'n thm
[WF(R), f ≡ WFREC R (λf x.M)] ⊢ f(pat_1) = rhs_1[(f | R, pat_1)]
[WF(R), f ≡ WFREC R (λf x.M)] ⊢ f(pat_n) = rhs_n[(f | R, pat_n)]
β-reduction; spec. patterns; case reduction
[WF(R), TC_11, ..., TC_1k_1, f ≡ WFREC R (λf x.M)] ⊢ f(pat_1) = rhs_1
[WF(R), TC_n1, ..., TC_nk_n, f ≡ WFREC R (λf x.M)] ⊢ f(pat_n) = rhs_n
termination condition extraction
Define function
f ≡ λX_1 ... X_j. WFREC (εR. WF(R) ∧ TC_11 ∧ ... ∧ TC_1k_1 ∧ TC_n1 ∧ ... ∧ TC_nk_n) (λf x.M)
R′
Eliminate definition; eliminate R′
Prove induction
[WF(R), TC_11, ..., TC_1k_1, ..., TC_n1, ..., TC_nk_n] ⊢ (f X_1 ... X_j (pat_1) = rhs_1[f X_1 ... X_j] ... f X_1 ... X_j (pat_n) = rhs_n[f X_1 ... X_j]) ∧ ind. thm

Figure 3.7: Schematic definition algorithm.

Eliminating the definition from the previous leaves

$$\begin{bmatrix} \mathsf{WF}\ (\varepsilon R.\ \mathsf{WF}\ R \land \forall s.\ B\ s \supset R\ (C\ s)\ s), \\ \forall s.\ B\ s \supset (\varepsilon R.\ \mathsf{WF}\ R \land \forall s.\ B\ s \supset R\ (C\ s)\ s)\ (C\ s)\ s \end{bmatrix}$$

$$\vdash$$

$$\mathsf{While}\ B\ C\ s = \mathtt{if}\ B\ s\ \mathtt{then}\ \mathsf{While}\ B\ C\ (C\ s)\ \mathtt{else}\ s.$$

Finally, assuming $\mathsf{WF}(R)$ and $\forall s.\ B\ s \supset R\ (C\ s)\ s$ and applying the Select Axiom allows the conclusion

$$\begin{bmatrix} \mathsf{WF}\ R, \\ \forall s.\ B\ s \supset R\ (C\ s)\ s \end{bmatrix}$$

$$\vdash$$

$$\mathsf{While}\ B\ C\ s = \mathtt{if}\ B\ s\ \mathtt{then}\ \mathsf{While}\ B\ C\ (C\ s)\ \mathtt{else}\ s.$$

□

The induction theorem that is derived from this definition is

$$\begin{bmatrix} \mathsf{WF}\ R, \\ \forall s.\ B\ s \supset R\ (C\ s)\ s \end{bmatrix} \vdash \forall P.\ (\forall s.\ (B\ s \supset P\ (C\ s)) \supset P\ s) \supset \forall v.\ P\ v.$$

If we define the semantics of a Hoare triple $\{P\}\ C\ \{Q\}$ as follows:

$$\mathsf{Hoare}\ (P, C, Q) \equiv \forall s.\ P\ s \supset Q\ (C\ s)$$

then it is a very simple application of the induction theorem to prove the following abstract While rule for total correctness:

$$\begin{bmatrix} \mathsf{WF}\ R, \\ \forall s.\ B\ s \supset R\ (C\ s)\ s \end{bmatrix} \vdash \begin{array}{l} \mathsf{Hoare}\ ((\boldsymbol{\lambda} s.\ P\ s \land B\ s),\ C,\ P)\ \supset \\ \mathsf{Hoare}\ (P, \mathsf{While}\ B\ C, (\boldsymbol{\lambda} s.\ P\ s \land \neg B\ s)). \end{array}$$

Remark. Had we tried to define While as a higher order function in the standard manner

$$\mathsf{While}\ B\ C\ s = \mathtt{if}\ B\ s\ \mathtt{then}\ \mathsf{While}\ B\ C\ (C\ s)\ \mathtt{else}\ s,$$

the extracted termination conditions

$$\mathsf{WF}\ R \land \forall B\ C\ s.\ B\ s \supset R\ (B, C, C\ s)\ (B, C, s).$$

would be unprovable (for example, C could be chosen to be the identity function and B could always return True, but there is no wellfounded relation R such that $R\ x\ x$). This gives some insight into our schematic definitions: they take a syntactic specification of a class of functions, *e.g.*, all while loops, and the use of wellfounded recursion for the semantics allows the total subset of that class to be captured in the logic.

Remark. This approach is not specific to wellfounded recursion: it should work for any fixpoint operator. In particular, for any fix satisfying the well-known fixpoint equation

$$\mathsf{fix}(M) = M\ (\mathsf{fix}(M)),$$

it is merely a common subexpression elimination to get

$$\forall g.\ g = \mathsf{fix}(M) \supset \forall x.\ g\ x = M\ g\ x.$$

In traditional applications of this theorem, M is a closed term. However, binding the free variables $\{X_1, \ldots X_j\}$ of M as parameters, we obtain:

$$(g = \boldsymbol{\lambda} X_1 \ldots X_j.\ \mathsf{fix}(M)) \supset \forall x.\ g\ X_1 \ldots X_j\ x = M\ (g\ X_1 \ldots X_j)\ x.$$

With hindsight, the treatment of parameters in inductive definition packages such as those reported in [72, 85, 48] can be seen as concrete applications of this theorem. In Section 5.6 we show how schematic definitions and schematic induction theorems can be used to derive program transformations.

3.7 Summary

In this chapter, we have shown how the basic logical tools supplied by the underlying proof system are put to work in defining functions and deriving induction schemes. Three techniques have been developed: the first requires the termination relation be supplied at definition time; the second makes the definition without a termination relation; and the third performs schematic definitions.

Our approach uses deductive steps to reduce the tasks to applications of the wellfounded recursion and induction theorems. There are other ways to define recursive functions, of course. Here we give a quick sketch of the ones known to us. A thorough comparison of the different possibilities would constitute valuable research.

1. Use Hilbert's choice operator to define $f(x) = M$ by $f \equiv \varepsilon f.\ \forall x.\ f(x) = M$. Making such a definition is effortless; however, this dodge may make reasoning with f more difficult than the other alternatives, since the ε operator is notoriously difficult to deal with mechanically (for example any nontrivial use immediately requires proving $\exists f.\ f(x) = M$). However, this approach may work well as a *front end*, treating other methods as means of proving the existence theorem.

2. Require all definitions to fit the Procrustean bed of higher order primitive recursion. This approach, although quite rich from the vantage point of proof theory, has little to recommend it for our target audience of functional programmers.

3. Inductively define the graph of the intended function, then prove that this graph is that of a function, then perform a type transformation from $\alpha \rightarrow \beta \rightarrow \mathsf{bool}$ to $\alpha \rightarrow \beta$. This approach is beguiling, since it seems to give a way of avoiding explicit consideration of wellfoundedness. The main question seems to be how hard the functionality proof is.

4. Interpret the functions as being over *domains* and give recursive definitions in terms of a fixpoint operator, instead of WFREC. This approach forces one to deal with two worlds of functions: normal functions and continuous functions. When the functions are total, as they often are, it seems like a large burden to maintain two worlds. It is an interesting problem to see how smooth passage back and forth between the worlds of total functions and continuous functions can be achieved. Müller [76] has made an extensive study of the topic.

5. Interpret functions as being set-theoretic least fixpoints. The paper [36] uses this approach to give a meta-theoretic description of a higher order logic similar to HOL. Least and greatest fixpoints have also been used *internally* in higher order logic for modelling recursive objects of all kinds [87, 77]. Least fixpoints do not have a natural notion of wellfoundedness, so termination proofs would require extra formalization.

Chapter 4

Nested and Mutual Recursion

In this chapter, we will explore two advanced kinds of recursion: nested and mutual. Both require more sophisticated treatment than that provided in the previous chapter, but they can, with some effort, be reduced to instances of wellfounded recursion and induction. It turns out that relationless and schematic versions of nested and mutual recursion can also be obtained.

4.1 Nested recursion

A function f is said to have a *nested* recursion when an argument to a recursive call of f contains another invocation of f. Nested recursion has traditionally posed problems and caused confusion. The intuitive reason for the difficulty is that, when a nested function is inductively constructed, each stage is not only allowed to refer to the function constructed at the previous stage, but also to *applications* of this function. For example, the main difficulty in establishing the totality of a nested function f is that the proof may rely on the value of an application of f. However, if f has not already been proved total, why should it be sound to use an application of f in the proof of totality? There seems to be a cyclic dependency between definition and valuation. This is such a serious problem that in some mechanized proof systems, *e.g.*, Nqthm, the totality of the function must be shown before the function can be defined. As a result, nested recursions in Nqthm must be defined indirectly. However, even if one can arrange that the function be defined before the proof of totality is carried out, as done in Chapter 3, nestedness may still cause trouble. For a running example, we use the following constant function that always returns zero:

$$\begin{array}{rcl}\mathsf{g}\,0 & \equiv & 0\\ \mathsf{g}\,(\mathsf{Suc}\,x) & \equiv & \mathsf{g}\,(\mathsf{g}\,x).\end{array}$$

No change to the definition algorithm is necessary to handle nested recursions, but we will step through it anyway. The pattern match translation yields:

$$\lambda g\ x.\ \mathsf{num\_case}\ 0\ (\lambda v.\ g(g\ v))\ x.$$

Definition. Invoke the primitive principle of definition. The termination relation is simply $<$.

$$\mathsf{g} \equiv \mathsf{WFREC}\ (<)\ (\lambda g\ x.\ \mathsf{num\_case}\ 0\ (\lambda v.\ g(g\ v))\ x).$$

Apply recursion theorem. By application of *modus ponens* with Theorem 24, the following theorems are derived (the wellfoundedness condition has been shunted to the assumptions):

$$\begin{array}{rcl} \mathsf{WF}(<) & \vdash & \forall x.\ \mathsf{g}\ x = \mathsf{num\_case}\ 0\ (\lambda v.\ (g \,|\, <, x)\,((\mathsf{g} \,|\, <, x)\ v)\ \ x. \\ & = & (\textit{instantiate patterns, reduce cases,} \\ & & \textit{eliminate wellfoundedness condition.}) \\ & \vdash & \mathsf{g}\ 0 = 0 \\ & \vdash & \mathsf{g}\ (\mathsf{Suc}\ x) = (\mathsf{g} \,|\, <, (\mathsf{Suc}\ x))\,((\mathsf{g} \,|\, <, (\mathsf{Suc}\ x))\ x)\,. \end{array}$$

Extract termination conditions. Two termination conditions are extracted for $\mathsf{g}(\mathsf{Suc}\ x)$. Notice that nestedness imposes a requirement on the traversal strategy of the extraction rewriter: termination conditions should be captured at innermost calls first. This can be accomplished via a bottom-up rewriting strategy, or by a top-down strategy that fails to match when the recursive call is nested.

$$\left[\begin{array}{c} x < \mathsf{Suc}\ x, \\ \mathsf{g}\ x < \mathsf{Suc}\ x \end{array}\right] \vdash \mathsf{g}(\mathsf{Suc}\ x) = \mathsf{g}(\mathsf{g}\ x).$$

Postprocessing. It is simple to prove the inner termination condition, leaving

$$[\mathsf{g}\ x < \mathsf{Suc}\ x] \vdash \mathsf{g}(\mathsf{Suc}\ x) = \mathsf{g}(\mathsf{g}\ x).$$

Proving the nested termination condition is not as simple. Consideration of this will be taken up in Section (4.1.2), after we discuss the derivation of induction.

4.1.1 Induction theorems

The style of induction theorems derived by TFL can be sloganized as '*the induction hypothesis holds for each argument to a recursive call*'. With this in mind, the following target induction theorem is desired for g:

$$\begin{array}{l} \forall P.\ P\,\mathsf{0}\ \wedge \\ \quad (\forall x.\ P\,x \wedge P\,(\mathsf{g}\,x) \supset P\,(\mathsf{Suc}\,x)) \\ \quad \supset \\ \quad \forall v.\ P\,v. \end{array}$$

Indeed, the algorithm of Section (3.4), will derive this induction theorem; however, the hypotheses will be the *fully* quantified termination conditions of g:

$$\begin{array}{l} \forall x.\ x < \mathsf{Suc}\,x, \\ \forall x.\ \mathsf{g}\,x < \mathsf{Suc}\,x \end{array}$$

Again, the first hypothesis is easy to eliminate. However, the second hypothesis is problematic. Attempting the proof by, say, mathematical induction doesn't get very far. A proof by complete induction works, but will require a case split and a lemma in order to obtain the required two inductive hypotheses. Attempting to prove the second hypothesis by use of the induction scheme itself isn't going to work, since there's a circularity problem: applying the scheme requires a proof of the original goal. However, a slight variant of the target induction scheme does work: instead of quantifying the nested termination condition and moving it onto the hypotheses, the conditions on the use of the nested induction hypothesis are left 'in place':

$$\left[\begin{array}{l} \mathsf{WF}\ (<), \\ \forall x.\ x < \mathsf{Suc}\,x \end{array}\right] \ \vdash\ \begin{array}{l} \forall P.\ P\,\mathsf{0}\ \wedge \\ \quad (\forall x.\ P\,x \wedge \underbrace{(\mathsf{g}\,x < \mathsf{Suc}\,x \supset P\,(\mathsf{g}\,x))} \supset P\,(\mathsf{Suc}\,x)) \\ \quad \supset \forall v.\ P\,v. \end{array} \tag{4.1}$$

Under this regime, a nested function definition results in an induction theorem with extra constraints on the use of inductive hypotheses for nested calls: to generalize from g a little, the input

$$\mathsf{f}\,x \equiv \ldots \mathsf{f}\,(\ldots \mathsf{f}\,(M\ x) \ldots) \ldots$$

along with a termination relation R results in the induction theorem

$$\left[\begin{array}{l} \mathsf{WF}\ R, \\ \forall x.\ R\,(M\ x)\ x \end{array}\right] \ \vdash\ \begin{array}{l} \forall P.\ (\forall x.\ P\,(M\ x)\ \wedge \\ \qquad\quad (R\,(\ldots \mathsf{f}\,(M\ x) \ldots)\ x \supset P\,(\ldots \mathsf{f}\,(M\ x) \ldots)) \\ \qquad\quad \supset P\,x) \\ \quad \supset \forall v.\ P\,v. \end{array}$$

Again, such theorems are difficult to understand and thus to work with. Therefore it is desirable for the termination constraints to be proved and eliminated before the induction theorem is used in later proofs.

4.1.2 Proving nested termination constraints

With a function with n levels of nested recursions, one eliminates the termination constraints by proceeding bottom up: first the non-nested constraints are proved and eliminated, then the constraints that have one level of nesting are proved and eliminated, and so on. But how are these nested constraints to be proved? It was previously held in the literature that nested termination constraints could only be proved by using the specification of the function, and thus that proofs of termination and correctness needed to be somehow intertwined. In fact, there are examples where this is not so: the function g is one, the $\mathsf{91}$ function is another, and the unification algorithm of Section 5.8 is another, much larger, example. In these examples, the nested termination condition can be proved by induction—sometimes even using the constrained induction scheme—making use of the ability to unroll the constrained function at smaller instances.

Example.

We assume that the constraints $\mathsf{WF}\ (<)$ and $\forall x.\ x < \mathsf{Suc}\ x$ have already been proved and eliminated from the assumptions of the recursion equations and the induction theorem for g. The nested termination constraint to be proved is

$$\forall x.\ \mathsf{g}\ x < \mathsf{Suc}\ x.$$

Proof. Induct with Theorem 4.1. This leaves two goals: $\mathsf{g}\ 0 < \mathsf{Suc}\ 0$, which is proved by unwinding the definition at the non-recursive clause, and the goal stemming from the recursive clause:

$$\begin{array}{l} \mathsf{g}\ (\mathsf{Suc}\ x) < \mathsf{Suc}\ (\mathsf{Suc}\ x) \\ \hline \quad 0.\ \ \mathsf{g}\ x < \mathsf{Suc}\ x \supset \mathsf{g}\ (\mathsf{g}\ x) < \mathsf{Suc}\ (\mathsf{g}\ x) \\ \quad 1.\ \ \mathsf{g}\ x < \mathsf{Suc}\ x \end{array}$$

Rewrite with the definition of g; this is allowed because the condition on unrolling g at $\mathsf{Suc}\ x$ is just assumption 1:

$$\begin{array}{l} \mathsf{g}\ (\mathsf{g}\ x) < \mathsf{Suc}\ (\mathsf{Suc}\ x) \\ \hline \quad 0.\ \ \mathsf{g}\ x < \mathsf{Suc}\ x \supset \mathsf{g}\ (\mathsf{g}\ x) < \mathsf{Suc}\ (\mathsf{g}\ x) \\ \quad 1.\ \ \mathsf{g}\ x < \mathsf{Suc}\ x \end{array}$$

The hypotheses yield $\mathsf{g}\ (\mathsf{g}\,x) < \mathsf{Suc}\ (\mathsf{g}\,x)$. The proof then completes by a chain of inequalities:

$$\mathsf{g}\ (\mathsf{g}\ x) \leq \mathsf{g}\ x < \mathsf{Suc}\ x < \mathsf{Suc}\ (\mathsf{Suc}\ x).$$

□

Now the recursion equations and induction scheme can be freed of the constraint, and can be applied to, *e.g.*, prove a specification of g:

$$\forall x.\ \mathsf{g}\ x = \mathsf{0}$$

Note that this property could also have been proven straightaway by use of the constrained recursion equations and mathematical induction. However, that doesn't invalidate our point: in many cases, termination and correctness can be proved separately. (This statement ignores, of course, the bald fact that termination is a kind of correctness property.)

4.2 Formal derivation of nested induction

To prove an induction theorem for a nested function, one has to make only minor modifications to the algorithm of Section 3.4. Essentially, these will ensure that the guard on the nested call is enforced: in order to use an inductive hypothesis at a nested call, the nested call must first be shown to be R-smaller than the pattern heading the clause. Steps 2{b,c} of the algorithm of Section 3.4 must be altered. Assume that we are enagaged in constructing the inductive hypothesis for a case of the recursive definition featuring a nested call:

$$\mathsf{f}\ (pat) \equiv \ldots\ \mathsf{f}\ (N\ (\mathsf{f}\ M))\ \ldots$$

Let the context of the inner call be $\Gamma(M)$, and that for the nested call be $\Gamma(N\ (\mathsf{f}\ M))$. The steps in the revised algorithm are as follows (we will only pay attention to the inner and nested calls) :

(2a). For any case pat_i, assume the instantiated antecedent of the wellfounded induction theorem (Theorem 6):

$$(a) \vdash \forall y.\ R\ y\ pat \supset P\ y.$$

(2b). Specialize (2a) to each recursive argument in the clause, getting

$$\begin{array}{lll} (a) & \vdash & R\ M\ pat \supset P\ M \\ (a) & \vdash & R\ (N\ (\mathsf{f}\ M))\ pat \supset P(N\ (\mathsf{f}\ M)). \end{array}$$

(2c). Prove each antecedent of the theorems from (2b). For the non-nested calls, proceed as before: namely, assume the fully quantified corresponding termination condition (and then specialize it)

$$\forall(\Gamma(M) \supset R\ M\ pat) \vdash \Gamma(M) \supset R\ M\ pat$$

then use transitivity of $\supset$ (or just *modus ponens*, when the context is empty), to eliminate $R\ M\ pat$ to get

$$(a), (\forall(\Gamma(M) \supset R\ M\ pat)) \vdash \Gamma(M) \supset P\ M.$$

For the nested calls, all that is necessary is to discharge the (non-existent) assumption $\Gamma(N\ (\mathsf{f}\ M))$ from (2b) (this requires the use of classical logic):

$$(a) \vdash \Gamma(N\ (\mathsf{f}\ M)) \supset R\ (N\ (\mathsf{f}\ M))\ pat \supset P(N\ (\mathsf{f}\ M)).$$

The rest of the algorithm proceeds without change. The main point is that the nested termination condition has not been moved to the assumptions, but is added to the context guard for the induction hypothesis for the nested call.

To sum up, the production of the recursion equations and induction theorem for a nested recursive function is not really any more difficult than in the non-nested case. However, it is often more involved to prove the nested termination conditions.

4.3 Relationless definition of nested functions

At first glance, it seems difficult to adapt the relationless definition method of Section 3.5 to nested functions. Conceptually, it seems hard to choose a suitable set of conditions for termination without mentioning the function being defined. (A secondary problem is that to unroll a nested function requires that the nested termination condition on the hypotheses be unquantified. However, the relationless definition approach of Section 3.5 universally quantifies the termination conditions in order to choose a closed relation, which is necessary to satisfy the principle of definition.)

In spite of these difficulties, in this section we will show that the definition of a nested recursive function *can* be separated from the delivery of its termination relation. The key idea is to proceed in two steps, combining the relationless definition approach of Section 3.5 with the ideas behind schematic definitions. Recall that the problem with defining a nested function in the relationless approach was that the function would appear in the termination constraints, and consequently the primitive principle of definition would fail. But we can make a schematic definition to handle this; after that, the relationless technique can be applied. We again proceed by example with the g function. Given a request to define

$$\begin{array}{rcl} \mathsf{g}\ 0 & \equiv & 0 \\ \mathsf{g}\ (\mathsf{Suc}\ x) & \equiv & \mathsf{g}\ (\mathsf{g}\ x), \end{array}$$

when a relation is not supplied, the first step is to compute the functional for the recursion equations, instantiate the recursion theorem, perform the 'case' reductions, and extract termination conditions:

$$\left[\begin{array}{l}\mathsf{WF}\ R,\\ \ G = \mathsf{WFREC}\ R\\ \quad (\boldsymbol{\lambda} G\ a.\ \mathsf{num\_case}\ 0\ (\boldsymbol{\lambda} v.\ G\ (G\ v))\ a)\end{array}\right] \vdash\ G\ 0 = 0$$

$$\left[\begin{array}{l}\mathsf{WF}\ R,\ R\ x\ (\mathsf{Suc}\ x),\ R\ (G\ x)\ (\mathsf{Suc}\ x),\\ \ G = \mathsf{WFREC}\ R\\ \quad (\boldsymbol{\lambda} G\ a.\ \mathsf{num\_case}\ 0\ (\boldsymbol{\lambda} v.\ G\ (G\ v))\ a)\end{array}\right] \vdash\ G\ (\mathsf{Suc}\ x) = G\ (G\ x)$$

Then the first definition is made; the auxiliary function is just parameterized by the termination relation:

$$\mathsf{aux}\ R \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} G\ a.\ \mathsf{num\_case}\ 0\ (\boldsymbol{\lambda} v.\ G\ (G\ v))\ a)$$

Then we proceed to cancel the definition from the previous theorem:

$$[\mathsf{WF}\ R]\ \vdash\ \mathsf{aux}\ R\ 0 = 0$$

$$\left[\begin{array}{l}\mathsf{WF}\ R,\\ R\ x\ (\mathsf{Suc}\ x),\\ R\ (\mathsf{aux}\ R\ x)\ (\mathsf{Suc}\ x)\end{array}\right] \vdash\ \mathsf{aux}\ R\ (\mathsf{Suc}\ x) = \mathsf{aux}\ R\ (\mathsf{aux}\ R\ x)$$

Now we make the second definition—the intended one—by gathering the termination conditions for the auxiliary function and choosing a relation satisfying them. Let εTC stand for $\varepsilon R.\ \mathsf{WF}\ R \wedge (\forall x.\ R\ x\ (\mathsf{Suc}\ x)) \wedge (\forall x.\ R\ (\mathsf{aux}\ R\ x)\ (\mathsf{Suc}\ x))$. Then define

$$\mathsf{g} \equiv \mathsf{aux}\ (\varepsilon TC)$$

and also prove, by the Select Axiom,

$$\begin{array}{l}[\mathsf{WF}\ R,\ (\forall x.\ R\ x\ (\mathsf{Suc}\ x)),\ (\forall x.\ R\ (\mathsf{aux}\ R\ x)\ (\mathsf{Suc}\ x))]\\ \quad\vdash\\ \mathsf{WF}\ (\varepsilon TC)\ \wedge\\ (\forall x.\ (\varepsilon TC)\ x\ (\mathsf{Suc}\ x))\ \wedge\\ (\forall x.\ (\varepsilon TC)\ (\mathsf{aux}\ (\varepsilon TC)\ x)\ (\mathsf{Suc}\ x))\end{array}$$

Substituting εTC for R in the equations for aux, we get

$$[\mathsf{WF}\ (\varepsilon TC)]\ \vdash\ \mathsf{aux}\ (\varepsilon TC)\ 0 = 0$$

$$\left[\begin{array}{l}\mathsf{WF}\ (\varepsilon TC),\\ (\varepsilon TC)\ x\ (\mathsf{Suc}\ x),\\ (\varepsilon TC)\ (\mathsf{aux}\ (\varepsilon TC)\ x)\ (\mathsf{Suc}\ x)\end{array}\right] \vdash\ \begin{array}{l}\mathsf{aux}\ (\varepsilon TC)\ (\mathsf{Suc}\ x)\\ \quad = \mathsf{aux}\ (\varepsilon TC)\ (\mathsf{aux}\ (\varepsilon TC)\ x)\end{array}$$

and finally,

$$\begin{array}{rcl} [\mathsf{WF}\ R] & \vdash & \mathsf{g}\ 0 = 0 \\ \left[\begin{array}{l} \mathsf{WF}\ R, \\ \forall x.\ R\ x\ (\mathsf{Suc}\ x), \\ \forall x.\ R\ (\mathsf{aux}\ R\ x)\ (\mathsf{Suc}\ x) \end{array}\right] & \vdash & \mathsf{g}\ (\mathsf{Suc}\ x) = \mathsf{g}\ (\mathsf{g}\ x). \end{array}$$

□

Looking at the result, one can see that there is a strict separation between assumptions, representing the termination conditions, and the conclusion, which delivers the requested function. Moreover, the choice of the termination condition is completely unconstrained in the result.

4.3.1 Formal derivation

We now give a more formal treatment of the algorithm.

1. Given a nested recursion,

$$\begin{array}{rcl} f(pat_1) & \equiv & rhs_1[f] \\ & \vdots & \\ f(pat_n) & \equiv & rhs_n[f] \end{array}$$

the machinery performs the same initial steps as a relationless definition, *viz.*, translates patterns, instantiates the recursion theorem, performs β-reduction, specializes the patterns, reduces the cases of the function, and finally performs termination condition extraction to arrive at

$$\begin{array}{lcl} \left[\begin{array}{l} \mathsf{WF}(R), TC_{11}[f], \ldots, TC_{1k_1}[f], \\ f \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M) \end{array}\right] & \vdash & f(pat_1) = rhs_1[f] \\ & \vdots & \\ \left[\begin{array}{l} \mathsf{WF}(R), TC_{n1}[f], \ldots, TC_{nk_n}[f], \\ f \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M) \end{array}\right] & \vdash & f(pat_n) = rhs_n[f]. \end{array}$$

Note that R is a variable not occurring in the original equations.

2. Now the definition of the auxiliary function is made:

$$\mathsf{aux}\ R \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M).$$

3. Immediately, the substitution $f \mapsto \textsf{aux}\ R$ can be made in the theorems from step 1, and the 'definitional assumption' can consequently be eliminated. Notice that the replacement of f takes place in the hypotheses as well, since the nested termination condition will be found there.

$$\begin{array}{rcl}
[\textsf{WF}(R), TC_{11}[\textsf{aux}\ R], \ldots, TC_{1k_1}[\textsf{aux}\ R]] & \vdash & \textsf{aux}\ R\ (pat_1) = rhs_1[\textsf{aux}\ R] \\
 & \vdots & \\
[\textsf{WF}(R), TC_{n1}[\textsf{aux}\ R], \ldots, TC_{nk_n}[\textsf{aux}\ R]] & \vdash & \textsf{aux}\ R\ (pat_n) = rhs_n[\textsf{aux}\ R]
\end{array}$$

4. Now consider a relation chosen to meet the termination conditions of $\textsf{aux}$:

$$\begin{array}{rll}
\varepsilon R.\ \textsf{WF}(R) & \wedge & \forall(TC_{11}[\textsf{aux}\ R]) \wedge \ldots \wedge \forall(TC_{1k_1}[\textsf{aux}\ R]) \\
 & \wedge & \forall(TC_{n1}[\textsf{aux}\ R]) \wedge \ldots \wedge \forall(TC_{nk_n}[\textsf{aux}\ R])
\end{array}$$

Call this term $\varepsilon R.TC$. Notice that this is a closed term. Now make the definition of the intended function:

$$\textsf{f} \equiv \textsf{aux}\ (\varepsilon R.TC).$$

What is now required is to bridge the gap between the auxiliary definition and the intended function.

5. Making the substitution $R \mapsto \varepsilon R.TC$ in the recursion equations for $\textsf{aux}$ from step 3 gives

$$\begin{array}{ccc}
\left[\begin{array}{l}
\textsf{WF}\ (\varepsilon R.TC), \\
{[R \mapsto \varepsilon R.TC]}\ (TC_{11}\ [\textsf{aux}\ R]), \ldots, \\
{[R \mapsto \varepsilon R.TC]}\ (TC_{1k_1}\ [\textsf{aux}\ R])
\end{array}\right] & \vdash & \begin{array}{c}
\textsf{aux}\ (\varepsilon R.TC)\ (pat_1) \\ = \\ {[R \mapsto \varepsilon R.TC]}\ (rhs_1[\textsf{aux}\ R])
\end{array} \\
 & \vdots & \\
\left[\begin{array}{l}
\textsf{WF}\ (\varepsilon R.TC), \\
{[R \mapsto \varepsilon R.TC]}\ (TC_{n1}\ [\textsf{aux}\ R]), \ldots, \\
{[R \mapsto \varepsilon R.TC]}\ (TC_{nk_n}\ [\textsf{aux}\ R])
\end{array}\right] & \vdash & \begin{array}{c}
\textsf{aux}\ (\varepsilon R.TC)\ (pat_n) \\ = \\ {[R \mapsto \varepsilon R.TC]}\ (rhs_n\ [\textsf{aux}\ R])
\end{array}
\end{array}$$

Since R did not occur in any of the original right hand sides, and also because the definition of $\textsf{f}$ has no free variables, it is valid to replace $\textsf{aux}\ (\varepsilon R.TC)$ by $\textsf{f}$ on the right hand sides of the previous theorem:

$$\begin{array}{ccc}
\left[\begin{array}{l}
\textsf{WF}\ (\varepsilon R.TC), \\
{[R \mapsto \varepsilon R.TC]}\ (TC_{11}\ [\textsf{aux}\ R]), \ldots, \\
{[R \mapsto \varepsilon R.TC]}\ (TC_{1k_1}\ [\textsf{aux}\ R])
\end{array}\right] & \vdash & \textsf{f}\ (pat_1) = rhs_1[\textsf{f}] \\
 & \vdots & \\
\left[\begin{array}{l}
\textsf{WF}\ (\varepsilon R.TC), \\
{[R \mapsto \varepsilon R.TC]}\ (TC_{n1}\ [\textsf{aux}\ R]), \ldots, \\
{[R \mapsto \varepsilon R.TC]}\ (TC_{nk_n}\ [\textsf{aux}\ R])
\end{array}\right] & \vdash & \textsf{f}\ (pat_n) = rhs_n[\textsf{f}]
\end{array}$$

It is important to abstain from performing this replacement in the assumptions.

6. All that is required now is to finesse the assumptions, and that can be achieved by use of the Select Axiom:

$$\begin{array}{l}[\mathsf{WF}(R), \forall(TC_{11}[\mathsf{aux}\ R]), \ldots, \forall(TC_{nk_n}[\mathsf{aux}\ R])] \\ \quad\vdash \\ \mathsf{WF}\ (\varepsilon R.TC) \land \\ \forall([R \mapsto \varepsilon R.TC](TC_{11}[\mathsf{aux}\ R])) \land \\ \vdots \\ \forall([R \mapsto \varepsilon R.TC](TC_{nk_n}[\mathsf{aux}\ R]))\end{array}$$

7. By invoking the Cut rule with (5) and (6), the final result is obtained:

$$\begin{array}{ll} \left[\begin{array}{l}\mathsf{WF}(R), \\ \forall(TC_{11}[\mathsf{aux}\ R]), \\ \quad\vdots \\ \forall(TC_{nk_n}[\mathsf{aux}\ R])\end{array}\right] & \vdash\ \mathsf{f}\ (pat_1) = rhs_1[\mathsf{f}] \\ & \quad\vdots \\ \left[\begin{array}{l}\mathsf{WF}(R), \\ \forall(TC_{11}[\mathsf{aux}\ R]), \\ \quad\vdots \\ \forall(TC_{nk_n}[\mathsf{aux}\ R])\end{array}\right] & \vdash\ \mathsf{f}\ (pat_n) = rhs_n[\mathsf{f}] \end{array}$$

□

Stepping back from the details, we have automatically derived the desired recursion equations, and generated an independent termination problem. Moreover, two means of settling the termination problem have also been derived fully automatically: the definition of the auxiliary function, and the induction scheme for the auxiliary function. The former will be of crucial importance in the proof, and the latter may also be required.

One might worry that the ability to prove termination has somehow been tampered with in the derivation. We believe that no termination arguments have been lost in this series of transformations.

Proof sketch. Suppose a function f is defined with the algorithm of Section 3.1, using a termination relation TR, *i.e.*,

$$\mathsf{f} \equiv \mathsf{WFREC}\ TR\ (\lambda f\ x.M).$$

Of the extracted termination conditions, the nested conditions will have occurrences of f. Suppose that the termination conditions are proved. In a relationless definition of f, the auxiliary function aux is defined:

$$\mathsf{aux}\,\mathcal{R} \equiv \mathsf{WFREC}\;\mathcal{R}\;(\boldsymbol{\lambda} f\;x.M).$$

Also, the same termination conditions are extracted as for f, except that occurrences of TR are instead a variable $\mathcal{R}$ and occurrences of f are instead applications $\mathsf{aux}\;\mathcal{R}$. If we substitute $\mathcal{R} \mapsto TR$ in the termination conditions, all the non-nested termination conditions are provable, since they are just the (non-nested) originals. That leaves the nested conditions, in which occurrences of $\mathsf{aux}\,\mathcal{R}$ are now $\mathsf{aux}\;TR$. It is trivial to show $\mathsf{f} = \mathsf{aux}\;TR$, and thus each nested termination condition is also provable.
□

4.3.2 Induction for relationless definition

It is simple to manipulate the induction theorem so that it is also separate from the termination problem. Returning to our running example, the induction theorem automatically generated for aux is

$$\begin{array}{l}
[\mathsf{WF}\;R, \forall x.\;R\;x\;(\mathsf{Suc}\;x)] \\
\quad\vdash \\
\forall P.\;\; P\;\mathsf{0} \land \\
\qquad (\forall x.\;P\;x \land (R\;(\mathsf{aux}\;R\;x)\;(\mathsf{Suc}\;x) \supset P\;(\mathsf{aux}\;R\;x)) \supset P\;(\mathsf{Suc}\;x)) \\
\qquad \supset \forall v.\;P\;v
\end{array}$$

First, notice that the nested termination condition is not an assumption, but is instead embedded in the conclusion of the theorem. Therefore, add the nested termination condition to the hypotheses, and then reduce the conclusion:

$$\begin{array}{l}
[\mathsf{WF}\;R,\;\forall x.\;R\;x\;(\mathsf{Suc}\;x),\;\forall x.\;R\;(\mathsf{aux}\;R\;x)(\mathsf{Suc}\;x)] \\
\quad\vdash \\
\forall P.\;\; P\;\mathsf{0} \land \\
\qquad (\forall x.\;P\;x \land P\;(\mathsf{aux}\;R\;x) \supset P\;(\mathsf{Suc}\;x)) \\
\qquad \supset \forall v.\;P\;v
\end{array}$$

As before, let εTC stand for

$$\varepsilon R.\;\mathsf{WF}\;R \land (\forall x.\;R\;x\;(\mathsf{Suc}\;x)) \land (\forall x.\;R\;(\mathsf{aux}\;R\;x)\;(\mathsf{Suc}\;x)).$$

Make the substitution $R \mapsto \varepsilon TC$ to obtain

$$\begin{array}{l}[\mathsf{WF}\ (\varepsilon TC), \forall x.\ (\varepsilon TC)\ x\ (\mathsf{Suc}\ x), \forall x.\ ((\varepsilon TC)\ (\mathsf{aux}\ (\varepsilon TC)\ x)\ (\mathsf{Suc}\ x)] \\ \quad \vdash \\ \forall P.\ P\ \mathsf{0} \land \\ \qquad (\forall x.\ P\ x \land P\ (\mathsf{aux}\ (\varepsilon TC)\ x) \supset P\ (\mathsf{Suc}\ x)) \\ \qquad \supset \forall v.\ P\ v\end{array}$$

By use of the Select Axiom, we can obtain

$$\begin{array}{l}[\mathsf{WF}\ R, \forall x.\ R\ x\ (\mathsf{Suc}\ x), \forall x.\ R\ (\mathsf{aux}\ R\ x)\ (\mathsf{Suc}\ x)] \\ \quad \vdash \\ \mathsf{WF}\ (\varepsilon TC) \land \\ \forall x.\ (\varepsilon TC)\ x\ (\mathsf{Suc}\ x) \land \\ \forall x.\ ((\varepsilon TC)\ (\mathsf{aux}\ (\varepsilon TC)\ x)\ (\mathsf{Suc}\ x)\end{array}$$

and thus

$$\begin{array}{l}[\mathsf{WF}\ R, \forall x.\ R\ x\ (\mathsf{Suc}\ x), \forall x.\ R\ (\mathsf{aux}\ R\ x)\ (\mathsf{Suc}\ x)] \\ \quad \vdash \\ \forall P.\ P\ \mathsf{0} \land \\ \qquad (\forall x.\ P\ x \land P\ (\mathsf{aux}\ (\varepsilon TC)\ x) \supset P\ (\mathsf{Suc}\ x)) \\ \qquad \supset \forall v.\ P\ v\end{array}$$

Finally, a simplification with the definition of g gives

$$\begin{array}{l}[\mathsf{WF}\ R, \forall x.\ R\ x\ (\mathsf{Suc}\ x), \forall x.\ R\ (\mathsf{aux}\ R\ x)\ (\mathsf{Suc}\ x)] \\ \quad \vdash \\ \forall P.\ P\ \mathsf{0} \land \\ \qquad (\forall x.\ P\ x \land P\ (\mathsf{g}\ x) \supset P\ (\mathsf{Suc}\ x)) \\ \qquad \supset \forall v.\ P\ v\end{array}$$

□

Algorithm

We now describe the formal derivation of induction theorems for nested relationless definitions. The following steps are taken:

1. Generate the induction theorem for the auxiliary function aux, by using the modification of Section (4.3.1). We will only focus on what happens at nested induction hypotheses, since the rest of the induction theorem will be unchanged by the following manipulations. Recall that a nested induction hypothesis is guarded by the context and the R-smaller condition:

$$\Gamma(N\ (\mathsf{aux}\ R\ M)) \supset R\ (N\ (\mathsf{aux}\ R\ M))\ \mathit{pat} \supset P\ (N\ (\mathsf{aux}\ R\ M)).$$

2. The assumptions of 1 consist of all the non-nested termination conditions. Add the nested termination conditions to the assumptions and then cancel them from the body of 1. Thus, the induction hypothesis is transformed to

   $$\Gamma(N\ (\mathsf{aux}\ R\ M)) \supset P(N\ (\mathsf{aux}\ R\ M)).$$

   and $\forall(R\ (N\ (\mathsf{aux}\ R\ M))\ pat)$ is added to the assumptions.

3. The hypotheses of 2 comprise the (quantified) termination conditions for the original function. Designate the conjunction of these requirements by TCs. Choose a relation satisfying TCs; call it εTCs. This is identical to the term from step 4 of Section (4.3.1).

4. Apply the substitution $R \mapsto \varepsilon TCs$ to (2). The only occurrences of R are in induction hypotheses for nested calls, and thus the substitutions in the body will only promote occurrences of $\mathsf{aux}\ R$ to $\mathsf{aux}\ \varepsilon TCs$. These new occurrences may now be replaced by f:

   $$\Gamma(N\ (\mathsf{f}\ M)) \supset P(N\ (\mathsf{f}\ M)).$$

   Again, it is important not to make such replacements in the assumptions.

5. Now the final step is to finesse the assumptions, exactly as in step 6 of (4.3.1).

The foundationally inclined reader may feel quite ill after this barrage of invocations of the Select Axiom. It would be interesting therefore, to try to find a way to make our definitions under the assumption that a satisfactory termination relation existed.

4.3.3 Example

McCarthy's 91 function:

$$\mathsf{91}\ x \equiv \texttt{if}\ x > 100\ \texttt{then}\ x - 10\ \texttt{else}\ \mathsf{91}(\mathsf{91}(x + 11))$$

is a venerable challenge problem. Defining it without a termination relation results in the following constrained equation and induction principle:

$$\begin{array}{l}
\left[\begin{array}{l}
\mathsf{WF}\ R, \\
\forall x.\ \neg(x > 100) \supset R\ (x+11)\ x, \\
\forall x.\ \neg(x > 100) \supset R\ (\mathsf{aux91}\ R\ (x+11))\ x
\end{array}\right] \\
\quad\ \vdash \\
(\mathsf{91}\ x = \mathtt{if}\ x > 100\ \mathtt{then}\ x - 10\ \mathtt{else}\ \mathsf{91}\ (\mathsf{91}\ (x+11))) \\
\wedge \\
(\forall P.\ (\forall x.\ (\neg(x > 100) \supset P\ (\mathsf{91}\ (x+11)))\wedge \\
\qquad\qquad\quad (\neg(x > 100) \supset P\ (x+11)) \supset P\ x) \\
\qquad\quad \supset \\
\qquad \forall v.\ P\ v).
\end{array} \tag{4.2}$$

The $\mathsf{91}$ function has the following charming property:

$$\forall x.\ \mathsf{91}\ x = \mathtt{if}\ x > 100\ \mathtt{then}\ x - 10\ \mathtt{else}\ 91,$$

which we will directly prove, before going on to consider termination. In the following, we will abuse notation and use $\mathsf{91}$ to denote the function, and 91 as a numeric literal.

Proof. Assume the termination constraints, then apply $\mathsf{91}$-induction. This gives the goal

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathsf{91}\ x = \mathtt{if}\ x > 100\ \mathtt{then}\ x - 10\ \mathtt{else}\ 91} \\
\hline
0. & \mathsf{WF}\ R \\
1. & \forall x.\ \neg(x > 100) \supset R\ (x+11)\ x \\
2. & \forall x.\ \neg(x > 100) \supset R\ (\mathsf{aux91}\ R\ (x+11))\ x \\
3. & \neg(x > 100) \supset \mathsf{91}\ (\mathsf{91}\ (x+11)) = \ \mathtt{if}\ \mathsf{91}\ (x+11) > 100 \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\ \ \mathtt{then}\ \mathsf{91}\ (x+11) - 10\ \mathtt{else}\ 91 \\
4. & \neg(x > 100) \supset \mathsf{91}\ (x+11) = \ \mathtt{if}\ x + 11 > 100 \\
 & \qquad\qquad\qquad\qquad\qquad\ \ \mathtt{then}\ (x+11) - 10\ \mathtt{else}\ 91.
\end{array}$$

Hypotheses 0, 1, and 2 are the termination constraints; we will omit to mention them in subsequent steps. Now the left hand side of the goal can be simplified with the definition of $\mathsf{91}$:

$$\begin{array}{c}
(\mathtt{if}\ x > 100\ \mathtt{then}\ x - 10\ \mathtt{else}\ \mathsf{91}\ (\mathsf{91}\ (x+11))) \\
= \\
(\mathtt{if}\ x > 100\ \mathtt{then}\ x - 10\ \mathtt{else}\ 91).
\end{array}$$

If $x > 100$, the result is trivial; thus assume $\neg(x > 100)$. It remains to show

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathsf{91}\ (\mathsf{91}\ (x+11)) = 91} \\
\hline
3. & \mathsf{91}(\mathsf{91}(x+11)) = \mathtt{if}\ \mathsf{91}(x+11) > 100\ \mathtt{then}\ \mathsf{91}(x+11) - 10\ \mathtt{else}\ 91 \\
4. & \mathsf{91}(x+11) = \mathtt{if}\ x + 11 > 100\ \mathtt{then}\ (x+11) - 10\ \mathtt{else}\ 91 \\
5. & \neg(x > 100)
\end{array}$$

Both induction hypotheses can be used at this point; first apply the nested hypothesis to obtain the goal

$$(\texttt{if}\ \mathsf{91}\ (x+11) > 100\ \texttt{then}\ \mathsf{91}\ (x+11) - 10\ \texttt{else}\ 91) = 91.$$

and then the non-nested hypothesis to obtain the goal

$$\left(\begin{array}{l} \texttt{if}\ \ (\texttt{if}\ x+11 > 100\ \texttt{then}\ (x+11) - 10\ \texttt{else}\ 91) > 100 \\ \texttt{then}\ (\texttt{if}\ x+11 > 100\ \texttt{then}\ (x+11) - 10\ \texttt{else}\ 91) - 10 \\ \texttt{else}\ 91 \end{array}\right) = 91.$$

Now the induction hypotheses can be thrown away; next, consider all the cases in the goal. Of the resulting goals, the only one that isn't immediate is the following:

$$\begin{array}{ll} \multicolumn{2}{l}{((x+11) - 10) - 10 = 91} \\ \hline 3. & \neg(x > 100) \\ 4. & x + 11 > 100 \\ 5. & (x+11) - 10 > 100 \\ 6. & \mathsf{91}\ (x+11) = (x+11) - 10 \end{array}$$

Hypotheses 3 and 5 imply that $x = 100$, and the result follows by calculation. □

The partial correctness of 91 has been proved; the resulting theorem is constrained by the assumed termination conditions (where the nested termination condition mentions aux91 instead of 91):

$$\begin{array}{l} \left[\begin{array}{l} \mathsf{WF}\ R, \\ \forall x.\ \neg(x > 100) \supset R\ (x+11)\ x, \\ \forall x.\ \neg(x > 100) \supset R\ (\mathsf{aux91}\ R\ (x+11))\ x \end{array}\right] \\ \quad \vdash \\ \forall x.\ \mathsf{91}\ x = \texttt{if}\ x > 100\ \texttt{then}\ x - 10\ \texttt{else}\ 91. \end{array} \tag{4.3}$$

Now we proceed to eliminate the termination conditions; the proof turns out to be more challenging than the correctness proof, but it can be made (indeed, it will be made) without reference to the partial correctness of 91. To get started, a termination relation for 91 (and thus aux91) must be selected; some thought shows that measure ($\boldsymbol{\lambda} x.\ 101{-}x$) is suitable. Thus, proving the following theorems will establish the termination of 91 (and thus allow the constraints on (4.3) to be lifted):

$$\begin{array}{l} \mathsf{WF}\ (\mathsf{measure}\ (\boldsymbol{\lambda} x.\ 101 - x)), \\ \forall x.\ \neg(x > 100) \supset \mathsf{measure}\ (\boldsymbol{\lambda} x.\ 101 - x)\ (x+11)\ x, \\ \forall x.\ \neg(x > 100) \supset \mathsf{measure}\ (\boldsymbol{\lambda} x.\ 101 - x) \\ \qquad\qquad\qquad\qquad (\mathsf{aux91}\ (\mathsf{measure}\ (\boldsymbol{\lambda} x.\ 101 - x))\ (x+11))\ x. \end{array}$$

The first two are easy to establish. To prove the third will require using the definition of aux91, which is just a parameterized version of 91:

$$\begin{array}{l}\left[\begin{array}{l}\mathsf{WF}\ R, \\ \forall x.\ \neg(x > 100) \supset R\ (x+11)\ x, \\ \forall x.\ \neg(x > 100) \supset R\ (\mathsf{aux91}\ R\ (x+11))\ x\end{array}\right] \\ \quad\vdash \\ \mathsf{aux91}\ R\ x = \begin{array}{ll}\mathtt{if}\ x > 100\ \mathtt{then} & x - 10 \\ \mathtt{else} & \mathsf{aux91}\ R\ (\mathsf{aux91}\ R\ (x+11))\end{array}\end{array} \tag{4.4}$$

At this point, for the sake of readability, we make the following abbreviation:

$$\mathcal{N} \equiv \mathsf{aux91}\ (\mathsf{measure}\ (\boldsymbol{\lambda} x.\ \mathsf{101} - x)).$$

Thus, instantiating R in (4.4) with $\mathsf{measure}\ (\boldsymbol{\lambda} x.\ \mathsf{101} - x)$ and eliminating the non-nested termination conditions gives :

$$\begin{array}{l}[\ \forall x.\ \neg(x > 100) \supset (\mathsf{measure}\ (\boldsymbol{\lambda} x.\ \mathsf{101} - x))\ (\mathcal{N}\ (x+11))\ x\ ] \\ \quad\vdash \\ \mathcal{N}\ x = \mathtt{if}\ x > 100\ \mathtt{then}\ x - 10\ \mathtt{else}\ \mathcal{N}\ (\mathcal{N}\ (x+11)).\end{array}$$

This can be recast as the following two equations:

$$\begin{array}{rcl}x > 100 & \vdash & \mathcal{N}\ x = x - 10 \\ \neg(x > 100),\ x < \mathcal{N}\ (x+11) & \vdash & \mathcal{N}\ x = \mathcal{N}\ (\mathcal{N}\ (x+11))\end{array} \tag{4.5}$$

The abbreviation also allows the nested condition to be rendered as:

$$\forall x.\ \neg(x > 100) \supset x < \mathcal{N}\ (x+11).$$

We now prove this. The proof will use (4.5): it may seem that there is a circularity, since one of the assumptions of (4.5) is just what is to be proved. Indeed, without discipline, that would be so; however, we will see how to unroll $\mathcal{N}$ such that the circularity is avoided.

Proof. This is a bounded quantification and can be proved by checking the results of evaluating $\mathcal{N}$ for each $n < 101$. We do not take this route. Alternatively, an induction theorem for aux91 has been derived. We will not use this either: it gives apparently hopeless goals. Instead, we will use complete induction on the termination relation. Applying this, and doing some simplification gives the following goal:

$$\begin{array}{ll}\multicolumn{2}{l}{x < \mathcal{N}\ (x+11)} \\ \hline 0. & \forall y.\ x < y \supset \neg(y > 100) \supset y < \mathcal{N}\ (y+11) \\ 1. & \neg(x > 100)\end{array}$$

The general idea of the proof is to progressively add conditions until the constraints on the recursion equations are satisfied, and the equations can be unrolled. Evidently, $\mathcal{N}$ must be unrolled at $x+11$. The clauses in the rewrite rules for $\mathcal{N}$ split on whether the input is greater than 100, so consider cases on whether $x+11>100$. If so, then we must show

$$\begin{array}{rcll} x & < & \mathcal{N}(x+11) & \\ & = & x+11-10 & \text{(unrolling } \mathcal{N}\text{)} \\ & = & x+1 & \end{array}$$

If not, assume $\neg(x+11>100)$. Since $x<x+11$, the induction hypothesis can be invoked to get $x+11<\mathcal{N}(x+22)$, or equivalently,

$$x<\mathcal{N}(x+22)-11.$$

Immediately try to invoke the inductive hypothesis on this: the first antecedent has just been created; if furthermore $\neg(\mathcal{N}(x+22)-11>100)$ holds, we obtain

$$\begin{array}{rcll} x<\mathcal{N}(x+22)-11 & < & \mathcal{N}((\mathcal{N}(x+22)-11)+11) & \text{(ind. hyp.)} \\ & = & \mathcal{N}(\mathcal{N}(x+22)) & \text{(arithmetic)} \\ & = & \mathcal{N}(x+11) & \text{(unrolling } \mathcal{N}\text{)} \end{array}$$

This would finish the proof. So how to show $\neg(\mathcal{N}(x+22)-11>100)$? It's easy: make a case split on $\mathcal{N}(x+22)-11>100$. The other part of the case split has the assumption

$$\begin{array}{lll} & \mathcal{N}(x+22)-11>100 & \\ \text{so} & \mathcal{N}(x+22)>100 & \text{(arithmetic)} \\ \text{so} & \mathcal{N}(\mathcal{N}(x+22))=\mathcal{N}(x+22)-10 & \text{(unrolling } \mathcal{N}\text{)} \\ \text{so} & \mathcal{N}(x+11)=\mathcal{N}(x+22)-10. & \text{(unrolling } \mathcal{N}\text{)} \end{array}$$

Now we are done, since

$$\begin{array}{rcll} x & < & \mathcal{N}(x+22)-11 & \\ & < & \mathcal{N}(x+22)-10 & \text{(because } \mathcal{N}(x+22)>100>11\text{)} \\ & = & \mathcal{N}(x+11). & \end{array}$$

□

In the proof, $\mathcal{N}$ is unrolled three times. Two of these use the base case:

argument	condition	result
$x+11$	$x+11>100$	$\mathcal{N}\ (x+11)=(x+11)-10=x+1$
$\mathcal{N}\ (x+22)$	$\mathcal{N}\ (x+22)>100$	$\mathcal{N}\ (\mathcal{N}\ (x+22))=\mathcal{N}\ (x+22)-10$

In the third case, one assumption is $\neg(x+11>100)$; the inductive hypothesis has been used to also prove $x+11<\mathcal{N}\ ((x+11)+11)$. Thus the hypotheses of the recursive clause of (4.5) are satisfied at the instance $x+11$, yielding $\mathcal{N}\ (x+11)=\mathcal{N}\ (\mathcal{N}\ (x+22))$. Therefore the last remaining termination condition of of $\mathcal{N}$

(and thus that of 91) has been established by using the recursion equations—in a non-circular manner. As a consequence, the constraints can be lifted from the definition of 91, the induction theorem, and the correctness theorem:

$$\vdash \mathsf{91}\ x = \mathtt{if}\ x > 100\ \mathtt{then}\ x - 10\ \mathtt{else}\ \mathsf{91}\ (\mathsf{91}\ (x + 11)),$$

$$\begin{aligned}\vdash \forall P.\ (\forall x.\ & (\neg(x > 100) \supset P\ (\mathsf{91}\ (x + 11)))\wedge \\ & (\neg(x > 100) \supset P\ (x + 11)) \supset P\ x) \\ \supset & \\ \forall v.\ & P\ v,\end{aligned}$$

$$\vdash \forall x.\ \mathsf{91}\ x = \mathtt{if}\ x > 100\ \mathtt{then}\ x - 10\ \mathtt{else}\ 91.$$

This example shows two things: first, a nested function can be defined and soundly reasoned about before its termination has been proved, indeed before its termination relation has even been formulated; second, the termination proof can—at least in this case—be performed in ignorance of the specification of the function. The termination proof of 91 only required the induction hypothesis, some basic arithmetic facts, and the ability to unroll the definition of aux91.

Since the functions 91 and aux91 are so similar, their relationship needs to be clear. The basic point is that the relationless definition of 91 is blocked by its nestedness. To get around this, an auxiliary function aux91 is defined; this definition is not hindered by its nestedness, because it is schematic in the termination relation. Now 91 can be defined as an application of aux91. The desired recursion equations and induction theorem of 91 can then be derived; these are constrained by the termination conditions of aux91. These termination conditions are provable if a wellfounded relation can be found that makes the termination conditions of 91 provable.

An alternative to having auxiliary functions would be to just use aux91 instead of 91. Although this makes good logical sense, it is less convincing methodologically. For example, if one wished to import a sequence of functional program definitions into TFL, a nested program would have to be translated to a parameterized function, and all subsequent programs would have to also be translated to use the parameterized version. Furthermore, any program using a nested program would have to likewise become parameterized by the termination relation. Although there would still be a one-to-one correspondence between program definitions and function definitions, the result could be a non-trivial distortion of the original.

A more attractive alternative to our approach would be if only one function was defined; however, the author has not been able to find such a scheme.

Finally, one might ask what happens if the termination relation was given at the time of defining 91, *i.e.*, if the algorithm of Section 3.1 was used. In that case, aux91 would not be defined, and every reference to $\mathcal{N}$ in the termination proof would instead be 91. The correctness proof could also be carried out before the termination proof, but it would be slightly more involved, since use of the nested induction hypothesis would first require a termination condition to be shown.

4.3.4 Nested schemes

As in Section (3.6), a little care must be taken with the parameters in schematic definitions, but otherwise, the definition algorithms and the derivation of induction are essentially unchanged for nested schemes. The only step requiring special treatment is the definition of the auxiliary function:

$$\mathsf{aux}\ R \equiv \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M).$$

This must now take account of $X_1, \ldots, X_k$, the free variables of $\boldsymbol{\lambda} f\ x.M$, as follows:

$$\mathsf{aux}\ R \equiv \boldsymbol{\lambda} X_1 \ldots X_k.\ \mathsf{WFREC}\ R\ (\boldsymbol{\lambda} f\ x.M).$$

4.4 Mutual recursion

Functions $f_1 \ldots f_k$ are *mutually* recursive when a call to some f_i results in a recursive call to a different f_j, and *vice versa*. To take a simple example, functions for deciding whether a number is even or odd can be elegantly written as the following mutual recursion:

$$\begin{aligned}
\mathsf{even}\ 0 &\equiv \mathsf{True} \\
\mathsf{even}\ (\mathsf{Suc}\ x) &\equiv \mathsf{odd}\ x \\[1em]
\mathsf{odd}\ 0 &\equiv \mathsf{False} \\
\mathsf{odd}\ (\mathsf{Suc}\ x) &\equiv \mathsf{even}\ x.
\end{aligned}$$

The definition of a collection of mutually recursive functions is typically handled by building a single ‘union’ function from which each individual function can be carved out. We shall use sum types to accomplish this. Thus, for even and odd, the union function EO : num + num → bool is defined (using the techniques from the previous chapter) as follows:

$$\begin{aligned}
\mathsf{EO}\ (\mathsf{INL}\ 0) &\equiv \mathsf{True} \\
\mathsf{EO}\ (\mathsf{INL}\ (\mathsf{Suc}\ x)) &\equiv \mathsf{EO}\ (\mathsf{INR}\ x) \\[1em]
\mathsf{EO}\ (\mathsf{INR}\ 0) &\equiv \mathsf{False} \\
\mathsf{EO}\ (\mathsf{INR}\ (\mathsf{Suc}\ x)) &\equiv \mathsf{EO}\ (\mathsf{INL}\ x).
\end{aligned}$$

Note, in particular, that nested patterns are required to support such definitions. Afterwards, the desired functions can be defined:

$$\begin{array}{rcl} \mathsf{even}\ x & \equiv & \mathsf{EO}\ (\mathsf{INL}\ x) \\ \mathsf{odd}\ x & \equiv & \mathsf{EO}\ (\mathsf{INR}\ x). \end{array}$$

Subsequently, the definitions of even and odd can be used (from right to left) to rewrite the definition of EO to get the specified recursion equations.

Turning to induction, we encounter a conceptual problem: it is not obvious what the induction theorem (theorems?) for even and odd should be. To pursue the approach already developed in this thesis, we want the induction theorem to reflect the recursive call structure of the specified functions. Existing methods for building induction schemes for mutual recursion either use the induction scheme for the union function [20], or expand out definitions in order to get induction schemes phrased solely in terms of the individual functions [56]. Instead, we have implemented a recent proposal from Richard Boulton [17], which fits neatly with our use of sums. He advocates the use of multiple induction predicates, one for each function. For example, the multi-predicate induction scheme for even and odd is:

$$\begin{array}{l} \forall P\ Q.\ P\ \mathsf{0} \wedge (\forall x.\ Q\ x \supset P\ (\mathsf{Suc}\ x)) \wedge \\ \qquad Q\ \mathsf{0} \wedge (\forall x.\ P\ x \supset Q\ (\mathsf{Suc}\ x)) \\ \qquad\quad \supset \\ \qquad (\forall v.\ P\ v) \wedge (\forall w.\ Q\ w). \end{array}$$

This theorem can be quite easily derived starting from the following induction scheme which has been automatically proved for EO:

$$\begin{array}{l} \forall P.\ P\ (\mathsf{INL}\ \mathsf{0}) \wedge \\ \quad (\forall x.\ P(\mathsf{INR}\ x) \supset P\ (\mathsf{INL}\ (\mathsf{Suc}\ x))) \wedge \\ \quad P\ (\mathsf{INR}\ \mathsf{0}) \wedge \\ \quad (\forall x.\ P(\mathsf{INL}\ x) \supset P\ (\mathsf{INR}\ (\mathsf{Suc}\ x))) \\ \qquad \supset \\ \quad \forall s.\ P\ s. \end{array}$$

The derivation starts by instantiating P to sum_case P Q. This delivers

$$\begin{array}{l} \mathsf{sum\_case}\ P\ Q\ (\mathsf{INL}\ \mathsf{0}) \wedge \\ (\forall x.\ \mathsf{sum\_case}\ P\ Q(\mathsf{INR}\ x) \supset \mathsf{sum\_case}\ P\ Q\ (\mathsf{INL}\ (\mathsf{Suc}\ x))) \wedge \\ \mathsf{sum\_case}\ P\ Q\ (\mathsf{INR}\ \mathsf{0}) \wedge \\ (\forall x.\ \mathsf{sum\_case}\ P\ Q(\mathsf{INL}\ x) \supset \mathsf{sum\_case}\ P\ Q\ (\mathsf{INR}\ (\mathsf{Suc}\ x))) \\ \quad \supset \\ \forall s.\ \mathsf{sum\_case}\ P\ Q\ s. \end{array}$$

Simplifying this with the definition of sum_case gives

$$\begin{array}{l} P\ 0 \land (\forall x.\ Q\ x \supset P\ (\mathsf{Suc}\ x)) \land \\ Q\ 0 \land (\forall x.\ P\ x \supset Q\ (\mathsf{Suc}\ x)) \\ \quad\supset \\ \forall s.\ \mathsf{sum\_case}\ P\ Q\ s. \end{array}$$

Now all that is necessary is to instantiate s, once with $\mathsf{INL}\ v$, and once with $\mathsf{INR}\ w$. Simplifying again with the definition of sum_case and then performing some trivial tidying-up steps gives the desired result.

Since the work of Gunter [46], multi-predicate induction schemes are commonly derived by mutually recursive datatype packages; what is new about Boulton's approach is that it works from recursion equations, and thus can derive induction for mutually recursive functions over a single datatype, as for even and odd.

Thus, it seems that the definition of mutually recursive functions can be reduced in a straightforward fashion to the definition of a single 'union' function, followed by the definition of the individual functions. There is only one further complication: in general, the 'union' function must be defined not only over the sum of the domain types of the individual functions, but also its range must be the sum of the *set* of range types of the individual functions.[1]

A larger example is used for illustration: evaluation functions over a mutually recursive datatype of first order arithmetic expressions. The component types comprise expressions (exp), and boolean expressions (bexp). An expression is a variable, a conditional, or an application of a function to a list of expressions. The 'test' of the conditional is a boolean expression, which may be an equality test, a test for 'less-than-or-equal', or a combination of boolean expressions (via NOT and OR).

$$\begin{array}{llcl}
\underline{\mathsf{exp}} & \mathsf{VAR} & : & \alpha \rightarrow (\alpha,\beta)\mathsf{exp} \\
 & \mathsf{IF} & : & (\alpha,\beta)\mathsf{bexp} \rightarrow (\alpha,\beta)\mathsf{exp} \rightarrow (\alpha,\beta)\mathsf{exp} \rightarrow (\alpha,\beta)\mathsf{exp} \\
 & \mathsf{APP} & : & \beta \rightarrow (\alpha,\beta)\mathsf{exp\ list} \rightarrow (\alpha,\beta)\mathsf{exp} \\
 & & & \\
\underline{\mathsf{bexp}} & \mathsf{EQ} & : & (\alpha,\beta)\mathsf{exp} \rightarrow (\alpha,\beta)\mathsf{exp} \rightarrow (\alpha,\beta)\mathsf{bexp} \\
 & \mathsf{LEQ} & : & (\alpha,\beta)\mathsf{exp} \rightarrow (\alpha,\beta)\mathsf{exp} \rightarrow (\alpha,\beta)\mathsf{bexp} \\
 & \mathsf{OR} & : & (\alpha,\beta)\mathsf{bexp} \rightarrow (\alpha,\beta)\mathsf{bexp} \rightarrow (\alpha,\beta)\mathsf{bexp} \\
 & \mathsf{NOT} & : & (\alpha,\beta)\mathsf{bexp} \rightarrow (\alpha,\beta)\mathsf{bexp}
\end{array}$$

The evaluation functions to be defined are straightforward; note that they are parameterized by a pair of environments Γ, which give bindings for variables and functions. The type of Γ, written $ty(\Gamma)$, is $(\alpha \rightarrow \mathsf{num})\#(\beta \rightarrow \mathsf{num\ list} \rightarrow \mathsf{num})$.

[1] Actually, this is an optimization: the sum of the range types would be satisfactory.

$$\begin{array}{lcl}
\mathsf{E}\ (\Gamma, \mathsf{VAR}\ x) & \equiv & \mathsf{fst}\ \Gamma\ x \\
\mathsf{E}\ (\Gamma, \mathsf{IF}\ b\ e_1\ e_2) & \equiv & \mathtt{if}\ \mathsf{EB}\ (\Gamma, b)\ \mathtt{then}\ \mathsf{E}\ (\Gamma, e_1)\ \mathtt{else}\ \mathsf{E}\ (\Gamma, e_2) \\
\mathsf{E}\ (\Gamma, \mathsf{APP}\ f\ l) & \equiv & (\mathsf{snd}\ \Gamma\ f)\ (\mathsf{EL}\ (\Gamma, l)) \\
 & & \\
\mathsf{EL}\ (\Gamma, [\,]) & \equiv & [\,] \\
\mathsf{EL}\ (\Gamma, h :: t) & \equiv & \mathsf{E}\ (\Gamma, h) :: \mathsf{EL}\ (\Gamma, t) \\
 & & \\
\mathsf{EB}\ (\Gamma, \mathsf{EQ}\ e_1\ e_2) & \equiv & \mathsf{E}\ (\Gamma, e_1) = \mathsf{E}\ (\Gamma, e_2) \\
\mathsf{EB}\ (\Gamma, \mathsf{LEQ}\ e_1\ e_2) & \equiv & \mathsf{E}\ (\Gamma, e_1) \leq \mathsf{E}\ (\Gamma, e_2) \\
\mathsf{EB}\ (\Gamma, \mathsf{NOT}\ b) & \equiv & \neg \mathsf{EB}\ (\Gamma, b) \\
\mathsf{EB}\ (\Gamma, \mathsf{OR}\ b_1\ b_2) & \equiv & \mathsf{EB}\ (\Gamma, b_1) \vee \mathsf{EB}\ (\Gamma, b_2)
\end{array}$$

The specified functions have the following types:

$$\begin{array}{lcl}
\mathsf{E} & : & ty(\Gamma)\#(\alpha, \beta)\mathsf{exp} \rightarrow \mathsf{num} \\
\mathsf{EL} & : & ty(\Gamma)\#(\alpha, \beta)\mathsf{exp\ list} \rightarrow \mathsf{num\ list} \\
\mathsf{EB} & : & ty(\Gamma)\#(\alpha, \beta)\mathsf{bexp} \rightarrow \mathsf{bool}.
\end{array}$$

Therefore, the union function $\boldsymbol{\mathcal{U}}$ will be typed as follows:

$$\boldsymbol{\mathcal{U}} : \left(\begin{array}{ll} (ty(\Gamma)\#(\alpha, \beta)\mathsf{exp}) & + \\ (ty(\Gamma)\#(\alpha, \beta)\mathsf{exp\ list}) & + \\ (ty(\Gamma)\#(\alpha, \beta)\mathsf{bexp}) & \end{array}\right) \longrightarrow \left(\begin{array}{ll} \mathsf{num} & + \\ \mathsf{num\ list} & + \\ \mathsf{bool} & \end{array}\right).$$

Convention. An n element sum can be written in a large number of ways; we will simply decree that the right-associated linear format

$$\tau_1 + \tau_2 + \ldots + \tau_{n-1} + \tau_n = \tau_1 + (\tau_2 + (\ldots + (\tau_{n-1} + \tau_n) \ldots))$$

is the one to adopt.

The description of the union function $\boldsymbol{\mathcal{U}}$ is built starting from the specified recursion equations. There are three considerations to address for each clause $f_i(pat) \equiv rhs$ in the original equations:

1. On the left hand side, the argument to f_i must be injected into the domain of $\boldsymbol{\mathcal{U}}$, *i.e.*, $f_i(pat)$ must be changed to $\boldsymbol{\mathcal{U}}\ (DOMINJ(f_i)(pat))$, where $DOMINJ(f_i)$ computes the injection for f_i. The following correspondences define $DOMINJ$ for the example:

$$\begin{array}{lcl}
\mathsf{E} & \mapsto & \boldsymbol{\lambda} x.\ \mathsf{INL}\ x \\
\mathsf{EL} & \mapsto & \boldsymbol{\lambda} x.\ \mathsf{INR}\ (\mathsf{INL}\ x) \\
\mathsf{EB} & \mapsto & \boldsymbol{\lambda} x.\ \mathsf{INR}\ (\mathsf{INR}\ x).
\end{array}$$

2. Each right hand side must be injected into the range of $\boldsymbol{\mathcal{U}}$, *i.e.*, *rhs* must be changed to *RNGINJ*(f_i)(*rhs*). The following correspondences define *RNGINJ* for the example:

$$\begin{array}{lcl} \mathsf{E} & \mapsto & \boldsymbol{\lambda} x.\mathsf{INL}\ x \\ \mathsf{EL} & \mapsto & \boldsymbol{\lambda} x.\mathsf{INR}\ (\mathsf{INL}\ x) \\ \mathsf{EB} & \mapsto & \boldsymbol{\lambda} x.\mathsf{INR}\ (\mathsf{INR}\ x). \end{array}$$

Note that the example under current consideration has the curious feature that *DOMINJ* and *RNGINJ* look the same; they aren't, since the types (not shown) are different. For comparison, recall even and odd: there *DOMINJ* is

$$\begin{array}{lcl} \mathsf{even} & \mapsto & \boldsymbol{\lambda} x.\ \mathsf{INL}\ x \\ \mathsf{odd} & \mapsto & \boldsymbol{\lambda} x.\ \mathsf{INR}\ x \end{array}$$

and *RNGINJ* is $\boldsymbol{\lambda} x.\ x$ for both even and odd.

3. Each occurrence of an f_k in *rhs* must be mapped to an application of $\boldsymbol{\mathcal{U}}$ to a suitably injected argument (via *DOMINJ*) *and then* projected out to the original range type of f_k. Thus a recursive call $f_k(x)$ will be translated to *RNGPROJ*$(f_k)(\boldsymbol{\mathcal{U}}$ (*DOMINJ*(f_k) (x))) where *RNGPROJ* is the compound projection function for the range of f_k. In the example, this is achieved by the following map:

$$\begin{array}{lcl} \mathsf{E} & \mapsto & \boldsymbol{\lambda} x.\ \mathsf{OUTL}\ x \\ \mathsf{EL} & \mapsto & \boldsymbol{\lambda} x.\ \mathsf{OUTL}\ (\mathsf{OUTR}\ x) \\ \mathsf{EB} & \mapsto & \boldsymbol{\lambda} x.\ \mathsf{OUTR}\ (\mathsf{OUTR}\ x). \end{array}$$

For example, taking the original clause

$$\mathsf{EL}\ (\Gamma, h :: t) \equiv \mathsf{E}\ (\Gamma, h) :: \mathsf{EL}\ (\Gamma, t),$$

the three transformation steps are as follows:

1. $\boldsymbol{\mathcal{U}}(\mathsf{INR}(\mathsf{INL}(\Gamma, h :: t))) \equiv \mathsf{E}\ (\Gamma, h) :: \mathsf{EL}\ (\Gamma, t)$
2. $\boldsymbol{\mathcal{U}}(\mathsf{INR}(\mathsf{INL}(\Gamma, h :: t))) \equiv \mathsf{INR}(\mathsf{INL}(\mathsf{E}\ (\Gamma, h) :: \mathsf{EL}\ (\Gamma, t)))$
3. $\boldsymbol{\mathcal{U}}(\mathsf{INR}(\mathsf{INL}(\Gamma, h :: t))) \equiv \mathsf{INR}(\mathsf{INL}\left(\begin{array}{l} \mathsf{OUTL}(\boldsymbol{\mathcal{U}}(\mathsf{INL}(\Gamma, h))) \\ :: \\ \mathsf{OUTL}(\mathsf{OUTR}(\boldsymbol{\mathcal{U}}(\mathsf{INR}(\mathsf{INL}(\Gamma, t))))) \end{array}\right))$

The final description of $\boldsymbol{\mathcal{U}}$ is the following:

$$
\begin{array}{lcl}
\mathcal{U}(\mathsf{INL}(\Gamma, \mathsf{VAR}\ x)) & \equiv & \mathsf{INL}(\mathsf{fst}\ \Gamma\ x) \\
\mathcal{U}(\mathsf{INL}(\Gamma, \mathsf{IF}\ b\ e_1\ e_2)) & \equiv & \mathsf{INL}\ \ (\mathtt{if}\ \mathsf{OUTR}(\mathsf{OUTR}(\mathcal{U}(\mathsf{INR}(\mathsf{INR}(\Gamma, b))))) \\
 & & \qquad \mathtt{then}\ \mathsf{OUTL}(\mathcal{U}(\mathsf{INL}(\Gamma, e_1))) \\
 & & \qquad \mathtt{else}\ \mathsf{OUTL}(\mathcal{U}(\mathsf{INL}(\Gamma, e_2)))) \\
\mathcal{U}(\mathsf{INL}(\Gamma, \mathsf{APP}\ f\ l)) & \equiv & \mathsf{INL}(\ (\mathsf{snd}\ \Gamma\ f) \\
 & & \qquad (\mathsf{OUTL}(\mathsf{OUTR}(\mathcal{U}(\mathsf{INR}(\mathsf{INL}(\Gamma, l))))))) \\
\mathcal{U}(\mathsf{INR}(\mathsf{INL}(\Gamma, []))) & \equiv & \mathsf{INR}(\mathsf{INL}[\,]) \\
\mathcal{U}(\mathsf{INR}(\mathsf{INL}(\Gamma, h :: t))) & \equiv & \mathsf{INR}(\mathsf{INL}\ (\mathsf{OUTL}(\mathcal{U}(\mathsf{INL}(\Gamma, h))) \\
 & & \qquad\qquad :: \\
 & & \qquad \mathsf{OUTL}(\mathsf{OUTR}(\mathcal{U}(\mathsf{INR}(\mathsf{INL}(\Gamma, t))))))) \\
\mathcal{U}(\mathsf{INR}(\mathsf{INR}(\Gamma, \mathsf{EQ}\, e_1 e_2))) & \equiv & \mathsf{INR}(\mathsf{INR}(\ \mathsf{OUTL}(\mathcal{U}(\mathsf{INL}(\Gamma, e_1))) \\
 & & \qquad\qquad = \\
 & & \qquad \mathsf{OUTL}(\mathcal{U}(\mathsf{INL}(\Gamma, e_2))))) \\
\mathcal{U}(\mathsf{INR}(\mathsf{INR}(\Gamma, \mathsf{LEQ}\, e_1 e_2))) & \equiv & \mathsf{INR}(\mathsf{INR}\ (\mathsf{OUTL}(\mathcal{U}(\mathsf{INL}(\Gamma, e_1))) \\
 & & \qquad\qquad \leq \\
 & & \qquad \mathsf{OUTL}(\mathcal{U}(\mathsf{INL}(\Gamma, e_2))))) \\
\mathcal{U}(\mathsf{INR}(\mathsf{INR}(\Gamma, \mathsf{NOT}\, b))) & \equiv & \mathsf{INR}(\mathsf{INR}(\neg(\mathsf{OUTR}(\mathsf{OUTR}(\mathcal{U}(\mathsf{INR}(\mathsf{INR}(\Gamma, b)))))))) \\
\mathcal{U}(\mathsf{INR}(\mathsf{INR}(\Gamma, \mathsf{OR}\, b_1 b_2))) & \equiv & \mathsf{INR}(\mathsf{INR}\ (\mathsf{OUTR}(\mathsf{OUTR}(\mathcal{U}(\mathsf{INR}(\mathsf{INR}(\Gamma, b_1))))) \\
 & & \qquad\qquad \vee \\
 & & \qquad \mathsf{OUTR}(\mathsf{OUTR}(\mathcal{U}(\mathsf{INR}(\mathsf{INR}(\Gamma, b_2))))))) \\
\end{array}
$$

This formula is input to the machinery discussed in this and the previous chapter; the function $\mathcal{U}$ is defined and the recursion equations are returned as a theorem, constrained by termination conditions. Also, a constrained induction theorem is returned for $\mathcal{U}$. Now the following definitions can be made:

$$
\begin{array}{lcl}
\mathsf{E}\ x & \equiv & \mathsf{OUTL}\ (\mathcal{U}\ (\mathsf{INL}\ x) \\
\mathsf{EL}\ x & \equiv & \mathsf{OUTL}\ (\mathsf{OUTR}\ (\mathcal{U}\ (\mathsf{INR}\ (\mathsf{INL}\ x)))) \\
\mathsf{EB}\ x & \equiv & \mathsf{OUTR}\ (\mathsf{OUTR}\ (\mathcal{U}\ (\mathsf{INR}\ (\mathsf{INR}\ x)))).
\end{array}
$$

These are used in the right-to-left orientation as rewrite rules, along with the laws for OUTL and OUTR, to simplify the definition of $\mathcal{U}$:

$$\begin{array}{lcl}
\boldsymbol{\mathcal{U}}(\mathsf{INL}(\Gamma, \mathsf{VAR}\ x)) & \equiv & \mathsf{INL}(\mathsf{fst}\ \Gamma\ x) \\
\boldsymbol{\mathcal{U}}(\mathsf{INL}(\Gamma, \mathsf{IF}\ b\ e_1\ e_2)) & \equiv & \mathsf{INL}(\mathtt{if}\ \mathsf{EB}(\Gamma, b)\ \mathtt{then}\ \mathsf{E}(\Gamma, e_1)\ \mathtt{else}\ \mathsf{E}(\Gamma, e_2)) \\
\boldsymbol{\mathcal{U}}(\mathsf{INL}(\Gamma, \mathsf{APP}\ f\ l)) & \equiv & \mathsf{INL}((\mathsf{snd}\ \Gamma\ f)\ (\mathsf{EL}\ (\Gamma, l))) \\
\boldsymbol{\mathcal{U}}(\mathsf{INR}(\mathsf{INL}(\Gamma, []))) & \equiv & \mathsf{INR}(\mathsf{INL}[]) \\
\boldsymbol{\mathcal{U}}(\mathsf{INR}(\mathsf{INL}(\Gamma, h :: t))) & \equiv & \mathsf{INR}(\mathsf{INL}(\mathsf{E}\ (\Gamma, h) :: \mathsf{EL}\ (\Gamma, t))) \\
\boldsymbol{\mathcal{U}}(\mathsf{INR}(\mathsf{INR}(\Gamma, \mathsf{EQ}\, e_1 e_2))) & \equiv & \mathsf{INR}(\mathsf{INR}(\mathsf{E}\ (\Gamma, e_1) = \mathsf{E}\ (\Gamma, e_2))) \\
\boldsymbol{\mathcal{U}}(\mathsf{INR}(\mathsf{INR}(\Gamma, \mathsf{LEQ}\, e_1 e_2))) & \equiv & \mathsf{INR}(\mathsf{INR}(\mathsf{E}\ (\Gamma, e_1) \leq \mathsf{E}\ (\Gamma, e_2))) \\
\boldsymbol{\mathcal{U}}(\mathsf{INR}(\mathsf{INR}(\Gamma, \mathsf{NOT}\, b))) & \equiv & \mathsf{INR}(\mathsf{INR}(\neg(\mathsf{EB}\ (\Gamma, b)))) \\
\boldsymbol{\mathcal{U}}(\mathsf{INR}(\mathsf{INR}(\Gamma, \mathsf{OR}\, b_1 b_2))) & \equiv & \mathsf{INR}(\mathsf{INR}(\mathsf{EB}\ (\Gamma, b_1) \vee \mathsf{EB}\ (\Gamma, b_2)))
\end{array}$$

The value of *RNGPROJ* for each function can now be applied on both sides of the equality; thus our example clause (now a theorem)

$$\boldsymbol{\mathcal{U}}(\mathsf{INR}(\mathsf{INL}(\Gamma, h :: t))) \equiv \mathsf{INR}(\mathsf{INL}(\mathsf{E}\ (\Gamma, h) :: \mathsf{EL}\ (\Gamma, t)))$$

is transformed via applying $\boldsymbol{\lambda} x.\mathsf{OUTL}(\mathsf{OUTR}\ x)$ to both sides to obtain

$$\mathsf{OUTL}(\mathsf{OUTR}(\boldsymbol{\mathcal{U}}(\mathsf{INR}(\mathsf{INL}(\Gamma, h :: t))))) \equiv \mathsf{OUTL}(\mathsf{OUTR}(\mathsf{INR}(\mathsf{INL}(\mathsf{E}(\Gamma, h) :: \mathsf{EL}(\Gamma, t))))).$$

The left hand side of this is an instance of EL, and the right hand side simplifies using the laws for OUTL and OUTR to deliver the initially requested recursion equation:

$$\mathsf{EL}\ (\Gamma, h :: t) \equiv \mathsf{E}\ (\Gamma, h) :: \mathsf{EL}\ (\Gamma, t).$$

That finishes the production of the recursion equations. The induction theorem derived for $\boldsymbol{\mathcal{U}}$ is the following:

$$\begin{array}{lll}
\forall P. & & (\forall \Gamma\ x.\ P\ (\mathsf{INL}(\Gamma, \mathsf{VAR}\ x))) \\
& \wedge & (\forall \Gamma\ b\ e_1 e_2.\ \ P\ (\mathsf{INR}(\mathsf{INR}(\Gamma, b)))\ \wedge \\
& & \qquad\qquad (\mathsf{EB}(\Gamma, b) \supset P\ (\mathsf{INL}(\Gamma, e_1)))\ \wedge \\
& & \qquad\qquad (\neg(\mathsf{EB}(\Gamma, b)) \supset P\ (\mathsf{INL}(\Gamma, e_2))) \supset P\ (\mathsf{INL}(\Gamma, \mathsf{IF}\ b\ e_1 e_2))) \\
& \wedge & (\forall \Gamma\ f\ l.\ P\ (\mathsf{INR}(\mathsf{INL}(\Gamma, l))) \supset P\ (\mathsf{INL}(\Gamma, \mathsf{APP}\ f\ l))) \\
& \wedge & (\forall \Gamma.\ P\ (\mathsf{INR}(\mathsf{INL}(\Gamma, [])))) \\
& \wedge & (\forall \Gamma\ h\ t.\ P\ (\mathsf{INL}(\Gamma, h)) \wedge P\ (\mathsf{INR}(\mathsf{INL}(\Gamma, t))) \supset P\ (\mathsf{INR}(\mathsf{INL}(\Gamma, h :: t)))) \\
& \wedge & (\forall \Gamma\ e_1 e_2.\ P\ (\mathsf{INL}(\Gamma, e_1)) \wedge P\ (\mathsf{INL}(\Gamma, e_2)) \supset P\ (\mathsf{INR}(\mathsf{INR}(\Gamma, \mathsf{EQ}\ e_1 e_2)))) \\
& \wedge & (\forall \Gamma\ e_1 e_2.\ P\ (\mathsf{INL}(\Gamma, e_1)) \wedge P\ (\mathsf{INL}(\Gamma, e_2)) \supset P\ (\mathsf{INR}(\mathsf{INR}(\Gamma, \mathsf{LEQ}\ e_1 e_2)))) \\
& \wedge & (\forall \Gamma\ b.\ P\ (\mathsf{INR}(\mathsf{INR}(\Gamma, b))) \supset P\ (\mathsf{INR}(\mathsf{INR}(\Gamma, \mathsf{NOT}\ b)))) \\
& \wedge & (\forall \Gamma\ b_1 b_2.\ P\ (\mathsf{INR}(\mathsf{INR}(\Gamma, b_2))) \wedge P\ (\mathsf{INR}(\mathsf{INR}(\Gamma, b_1))) \\
& & \qquad\qquad \supset P\ (\mathsf{INR}(\mathsf{INR}(\Gamma, \mathsf{OR}\ b_1 b_2)))) \\
& \supset & \\
& & \forall v.\ P\ v.
\end{array}$$

Now 3 induction predicates are created:

$$\begin{array}{lcl} P_1 & : & ty(\Gamma)\#(\alpha,\beta)\mathsf{exp} \rightarrow \mathsf{bool} \\ P_2 & : & ty(\Gamma)\#(\alpha,\beta)\mathsf{exp\ list} \rightarrow \mathsf{bool} \\ P_3 & : & ty(\Gamma)\#(\alpha,\beta)\mathsf{bexp} \rightarrow \mathsf{bool} \end{array}$$

and the sum of these, $\mathsf{sum\_case}\ P_1\ (\mathsf{sum\_case}\ P_2\ P_3)$, is used as an instantiation for P in the induction theorem. For example, the induction clause

$$\forall \Gamma\ h\ t.\ P\ (\mathsf{INL}(\Gamma, h)) \wedge P\ (\mathsf{INR}(\mathsf{INL}(\Gamma, t))) \supset P\ (\mathsf{INR}(\mathsf{INL}(\Gamma, h :: t)))$$

is instantiated to

$$\begin{array}{ll} \forall \Gamma\ h\ t. & (\mathsf{sum\_case}\ P_1\ (\mathsf{sum\_case}\ P_2\ P_3))\ (\mathsf{INL}(\Gamma, h)) \wedge \\ & (\mathsf{sum\_case}\ P_1\ (\mathsf{sum\_case}\ P_2\ P_3))\ (\mathsf{INR}(\mathsf{INL}(\Gamma, t))) \\ & \quad\supset \\ & (\mathsf{sum\_case}\ P_1\ (\mathsf{sum\_case}\ P_2\ P_3))\ (\mathsf{INR}(\mathsf{INL}(\Gamma, h :: t))) \end{array}$$

Simplification with the definition of $\mathsf{sum\_case}$ then leaves

$$\forall \Gamma\ h\ t. P_1\ (\Gamma, h) \wedge P_2\ (\Gamma, t) \supset P_2\ (\Gamma, h :: t).$$

As a result, the induction theorem has now been transformed to

$$\boxed{\begin{array}{ll} & (\forall \Gamma\ x.\ P_1\ (\Gamma, \mathsf{VAR}\ x))) \\ \wedge & (\forall \Gamma\ b\ e_1 e_2.\ \ P_3\ (\Gamma, b) \wedge \\ & \qquad\qquad\quad (\mathsf{EB}(\Gamma, b) \supset P_1\ (\Gamma, e_1)) \wedge \\ & \qquad\qquad\quad (\neg(\mathsf{EB}(\Gamma, b)) \supset P_1\ (\Gamma, e_2)) \supset P_1\ (\Gamma, \mathsf{IF}\ b\ e_1 e_2)) \\ \wedge & (\forall \Gamma\ f\ l.\ P_2\ (\Gamma, l) \supset P_1\ (\Gamma, \mathsf{APP}\ f\ l) \\ \wedge & (\forall \Gamma.\ P_2\ (\Gamma, [])) \\ \wedge & (\forall \Gamma\ h\ t.\ P_1\ (\Gamma, h) \wedge P_2\ (\Gamma, t) \supset P_2\ (\Gamma, h :: t)) \\ \wedge & (\forall \Gamma\ e_1 e_2.\ P_1\ (\Gamma, e_1) \wedge P_1\ (\Gamma, e_2) \supset P_3\ (\Gamma, \mathsf{EQ}\ e_1 e_2)) \\ \wedge & (\forall \Gamma\ e_1 e_2.\ P_1\ (\Gamma, e_1) \wedge P_1\ (\Gamma, e_2) \supset P_3\ (\Gamma, \mathsf{LEQ}\ e_1 e_2)) \\ \wedge & (\forall \Gamma\ b.\ P_3\ (\Gamma, b) \supset P_3\ (\Gamma, \mathsf{NOT}\ b)) \\ \wedge & (\forall \Gamma\ b_1 b_2.\ P_3\ (\Gamma, b_1) \wedge P_3\ (\Gamma, b_2) \supset P_3\ (\Gamma, \mathsf{OR}\ b_1 b_2)) \\ \supset & \\ & \forall v.\ \mathsf{sum\_case}\ P_1\ (\mathsf{sum\_case}\ P_2\ P_3)\ v. \end{array}}$$

The consequent of this theorem is now instantiated three times using *DOMINJ*: v takes on the values of $\mathsf{INL}\ x$, $\mathsf{INR}\ (\mathsf{INL}\ y)$, and $\mathsf{INR}\ (\mathsf{INR}\ z)$. This results in the three consequents

$$\begin{array}{l}\mathsf{sum\_case}\ P_1\ (\mathsf{sum\_case}\ P_2\ P_3)\ (\mathsf{INL}\ x)\\ \mathsf{sum\_case}\ P_1\ (\mathsf{sum\_case}\ P_2\ P_3)\ (\mathsf{INR}\ (\mathsf{INL}\ y))\\ \mathsf{sum\_case}\ P_1\ (\mathsf{sum\_case}\ P_2\ P_3)\ (\mathsf{INR}\ (\mathsf{INR}\ z)).\end{array}$$

Simplification with the definition of sum_case gives the consequents $P_1\ x$, $P_2\ y$, and $P_3\ z$. These can be universally quantified with respect to x, y, z, rendering the final induction theorem:

$$\boxed{\begin{array}{ll} & (\forall \Gamma\ x.\ P_1\ (\Gamma, \mathsf{VAR}\ x)))\\ \wedge & (\forall \Gamma\ b\ e_1 e_2.\ \ P_3\ (\Gamma, b)\ \wedge\\ & \qquad\qquad (\mathsf{EB}(\Gamma, b) \supset P_1\ (\Gamma, e_1))\ \wedge\\ & \qquad\qquad (\neg(\mathsf{EB}(\Gamma, b)) \supset P_1\ (\Gamma, e_2)) \supset P_1\ (\Gamma, \mathsf{IF}\ b\ e_1 e_2))\\ \wedge & (\forall \Gamma\ f\ l.\ P_2\ (\Gamma, l) \supset P_1\ (\Gamma, \mathsf{APP}\ f\ l)\\ \wedge & (\forall \Gamma.\ P_2\ (\Gamma, []))\\ \wedge & (\forall \Gamma\ h\ t.\ P_1\ (\Gamma, h) \wedge P_2\ (\Gamma, t) \supset P_2\ (\Gamma, h :: t))\\ \wedge & (\forall \Gamma\ e_1 e_2.\ P_1\ (\Gamma, e_1) \wedge P_1\ (\Gamma, e_2) \supset P_3\ (\Gamma, \mathsf{EQ}\ e_1 e_2))\\ \wedge & (\forall \Gamma\ e_1 e_2.\ P_1\ (\Gamma, e_1) \wedge P_1\ (\Gamma, e_2) \supset P_3\ (\Gamma, \mathsf{LEQ}\ e_1 e_2))\\ \wedge & (\forall \Gamma\ b.\ P_3\ (\Gamma, b) \supset P_3\ (\Gamma, \mathsf{NOT}\ b))\\ \wedge & (\forall \Gamma\ b_1 b_2.\ P_3\ (\Gamma, b_1) \wedge P_3\ (\Gamma, b_2) \supset P_3\ (\Gamma, \mathsf{OR}\ b_1 b_2))\\ \supset & \\ & (\forall x.\ P_1\ x) \wedge (\forall y.\ P_2\ y) \wedge (\forall z.\ P_3\ z).\end{array}}$$

4.4.1 Formal derivation of mutual recursion

Given mutually recursive equations

$$\begin{array}{rcl} f_1(pat_{11}) & \equiv & rhs_{11}\\ & \vdots & \\ f_n(pat_{nk_n}) & \equiv & rhs_{nk_n}\end{array}$$

the definition algorithm proceeds as follows.

1. Compute domain and range sums. The types of the specified functions

$$\begin{array}{rcl} f_1 & : & \sigma_1 \to \tau_1\\ & \vdots & \\ f_n & : & \sigma_n \to \tau_n\end{array}$$

lead to the domain type $domty \equiv \sigma_1 + \ldots + \sigma_n$. Let $\rho_1 \ldots \rho_j$ enumerate the set $\{\tau_1, \ldots, \tau_n\}$. Then $rngty \equiv \rho_1 + \ldots + \rho_j$. The union function will have type $domty \to rngty$.

2. **Compute injection and projection maps.** Construct maps from the individual functions to injection and projection functions for *domty* and *rngty*.

$$\begin{array}{rcll} \mathit{DOMINJ} & \equiv & \{f_i \mapsto \mathsf{IN}(i-1)(\mathit{domty}) & \mid\ i \in 1 \ldots n\} \\ \mathit{RNGINJ} & \equiv & \{f_i \mapsto \mathsf{IN}(k-1)(\mathit{rngty}) & \mid\ i \in 1 \ldots n \wedge \rho_k = \tau_i\} \\ \mathit{RNGPROJ} & \equiv & \{f_i \mapsto \mathsf{OUT}(k-1)(\mathit{rngty}) & \mid\ i \in 1 \ldots n \wedge \rho_k = \tau_i\} \end{array}$$

Defining the auxiliary functions IN and OUT is an interesting exercise in functional programming; the following can perhaps be better expressed using streams:

$$\begin{array}{rcl} \mathsf{IN}\ m\ (\gamma_0 + \ldots + \gamma_k) & \equiv & \mathsf{IN}_0\ (m,\ \mathsf{0},\ \boldsymbol{\lambda} x : \gamma_0 + \ldots + \gamma_k.\ x) \\ \mathsf{IN}_0\ (\mathsf{0},\ i,\ f) & \equiv & \mathtt{if}\ i = k\ \mathtt{then}\ f\ \mathtt{else}\ f \circ \mathsf{INL} \\ \mathsf{IN}_0\ (\mathsf{Suc}\ m,\ i,\ f) & \equiv & \mathsf{IN}_0\ (m,\ \mathsf{Suc}\ i,\ f \circ \mathsf{INR}) \end{array}$$

$$\begin{array}{rcl} \mathsf{OUT}\ m\ (\gamma_0 + \ldots + \gamma_k) & \equiv & \mathsf{OUT}_0\ (m,\ \mathsf{0},\ \boldsymbol{\lambda} x : \gamma_0 + \ldots + \gamma_k.\ x) \\ \mathsf{OUT}_0\ (\mathsf{0},\ i,\ f) & \equiv & \mathtt{if}\ i = k\ \mathtt{then}\ f\ \mathtt{else}\ \mathsf{OUTL} \circ f \\ \mathsf{OUT}_0\ (\mathsf{Suc}\ m,\ i,\ f) & \equiv & \mathsf{OUT}_0\ (m,\ \mathsf{Suc}\ i,\ \mathsf{OUTR} \circ f) \end{array}$$

Remark. One might notice that *DOMPROJ*, *i.e.*, the projections for *domty*, is missing: it is subsumed by the pattern-matching translation applied when the union function is defined in step 4.

3. **Apply transformations.** Make $\mathcal{U}$: *domty* → *rngty*. For each $f_i(\mathit{pat}) = \mathit{rhs}$ clause, do the following:

   1. Replace $f_i(\mathit{pat})$ by $\mathcal{U}\ (\mathit{DOMINJ}\ f_i\ \mathit{pat})$.
   2. Replace occurrences of each f_k by $\mathit{RNGPROJ}\ f_k \circ \mathcal{U} \circ \mathit{DOMINJ}\ f_k$ in rhs. Call this rhs'.
   3. Inject rhs' into the range sum. Thus the final equation is

$$\mathcal{U}\ (\mathit{DOMINJ}\ f_i\ \mathit{pat}) \equiv \mathit{RNGINJ}\ f_i\ \mathit{rhs}'.$$

This is well-typed if the original equation is, although to prove it formally would seem to take a lot of work. β-conversion and simplification with the definition of $\circ$ should now be applied.

4. **Define union function.** If a relation has been supplied, define $\mathcal{U}$ with the algorithm of Section 3.1. If no relation has been supplied and $\mathcal{U}$ is not nested, invoke the algorithms from section 3.5; otherwise, if no relation has been supplied and $\mathcal{U}$ is nested, invoke the algorithms from Section 4.3.1. The result is a conjunction of recursion equations and an induction theorem.

5. **Define individual functions.** For each f_i, make the following definition:

$$\mathsf{f}_i \equiv \mathit{RNGPROJ}\ f_i \circ \boldsymbol{\mathcal{U}} \circ \mathit{DOMINJ}\ f_i.$$

6. **Derive original equations.** For each equation resulting from step 4, apply *RNGPROJ* f_i to both sides of the equality. Simplify this with the definitions of step 5, using the definitions in the right-to-left orientation; also apply the theorems $\vdash$ OUTL (INL x) $= x$ and $\vdash$ OUTR (INR x) $= x$.

4.4.2 Formal derivation of mutual induction

The induction theorem for $\boldsymbol{\mathcal{U}}$ from step 4 of the definition algorithm has the form $\forall P.$ *induction clauses* $\supset \forall x.\ P\ x$, where P has type *domty* $\rightarrow$ bool.

1. Create n new predicate variables $P_1 : \sigma_1 \rightarrow \mathsf{bool}, \ldots, P_n : \sigma_n \rightarrow \mathsf{bool}$. Instantiate P in the induction theorem by sum_case $P_1 \ldots$ (sum_case $P_{n-1}\ P_n$). Simplify the result with the definition of sum_case.

2. Make n instantiations of x in the theorem coming from step 1:

$$\begin{array}{lcl} x & \mapsto & \mathit{DOMINJ}(f_1) \\ & \vdots & \\ x & \mapsto & \mathit{DOMINJ}(f_n). \end{array}$$

   Simplify each result with the definition of sum_case. This results in n theorems of the form

$$\begin{array}{rcl} \mathit{induction\ clauses} & \supset & P_1\ x_1 \\ & \vdots & \\ \mathit{induction\ clauses} & \supset & P_n\ x_n. \end{array}$$

3. If desired, universally quantify each $x_1, \ldots, x_n$.

4. Conjoin the resulting theorems, obtaining a theorem of the form

$$\mathit{induction\ clauses} \supset (\forall x_1.\ P_1\ x_1) \wedge \ldots \wedge (\forall x_n.\ P_n\ x_n).$$

   If step 3 has been skipped, the result will look like

$$\mathit{induction\ clauses} \supset P_1\ x_1 \wedge \ldots \wedge P_n\ x_n.$$

5. If desired, universally quantify $P_1, \ldots, P_n$ in the result.

4.5 Related work

Boyer and Moore [20] require nested definitions to be first proved to satisfy non-nested recursion equations. They recommend that mutual recursion (and the induction scheme) be instead phrased in terms of what we have been calling the 'union' function.

PVS relies on its type system to support nested recursive definitions. Essentially, the specification of the function is used in proving termination: nested recursive calls are required to lie in the set of behaviours of the function by clever use of subtyping [79]. PVS does not appear to support mutual recursion currently.

LAMBDA defines nested and mutual recursive definitions via a fixpoint operator, but doesn't automatically derive induction theorems, although Busch shows how induction theorems can be manually derived in LAMBDA [47].

Giesl [38] also made the observation—independently but earlier—that termination and correctness need not be intertwined for nested functions. In [40], he shows that, if nested termination conditions can be proved by the specified induction theorem for a nested function, then such a proof is sound. In the same paper, he describes a powerful automated method for automatically proving termination of nested functions (it can prove the termination of the 91 function). Giesl's work is presented in the setting of first order logic and uses such notions as call-by-value evaluation on ground terms; in addition his theorems are justified meta-theoretically. In contrast, our definitions, being total functions in classical logic, are oblivious to evaluation strategy, and can moreover be higher order and schematic. Since our derivations all proceed by object-logic deduction in a sound logic, we need make no soundness argument.

Kapur and Subramaniam [56] show how the RRL proof system can use its cover set induction method to tackle the automation of mutual induction. Work related to this can be found in Spike [15].

Researchers in Type Theory have evolved several means of dealing with nested recursion; in early work, Nördstrom proposed *accessibility* relations to increase the power of Martin-Löf Type Theory so that it can express general recursions, including nested recursion[78].

Chapter 5

Examples

This chapter is devoted to examples that highlight our approach.

5.1 List permutations and sorting

Our aim in this section is to study some sorting algorithms. A sorted list must have all (and only) the elements in the original list, *i.e.*, be a permutation, and the elements must be in ascending (or descending) order according to some order relation. Higher order logic gives a simple definition of permutation (there are equally viable alternatives):

Definition 49 (perm)

$$\mathsf{perm}\ l_1 l_2 \equiv \forall x.\ \mathsf{filter}(\$=\ x) l_1 = \mathsf{filter}(\$=\ x) l_2$$

In this definition, \$ is used to defeat the infix parsing staus of equality (=), which is curried. Thus $\$=\ x$ is a function of type $\alpha \to \mathsf{bool}$. The following theorems about permutations are required in the verification of Quicksort. They are all quite simple to prove.

perm_refl	perm $l\ l$
perm_transitive	transitive perm
perm_sym	perm $l_1\ l_2 =$ perm $l_2\ l_1$
perm_cong	perm $l_1 l_3 \land$ perm $l_2 l_4 \supset$ perm$(l_1 \mathbin{@} l_2)(l_3 \mathbin{@} l_4)$
cons_perm	perm $l\ (M \mathbin{@} N) \supset$ perm $(x :: l)\ (M \mathbin{@} (x :: N))$
append_perm_sym	perm $(A \mathbin{@} B)\ C \supset$ perm $(B \mathbin{@} A)\ C$
perm_split	perm l (filter $P\ l \mathbin{@}$ filter $(\neg \circ P)\ l$)

Now we define what it means for an order R to hold pairwise throughout a list.

Definition 50 (sorted)

$$\begin{array}{rcl} \mathsf{sorted}(R,[\,]) & \equiv & \mathsf{True} \\ \mathsf{sorted}(R,[x]) & \equiv & \mathsf{True} \\ \mathsf{sorted}(R, x :: y :: rst) & \equiv & R\ x\ y \wedge \mathsf{sorted}(R, y :: rst). \end{array}$$

The following theorems about sorted are used in the correctness proofs:

sorted_eq	transitive R $\supset \forall x.\ \mathsf{sorted}(R, x :: l) = \mathsf{sorted}(R, l) \wedge \forall y.\ \mathsf{mem}\ y\ l \supset R\ x\ y$
sorted_append	transitive $R \wedge \mathsf{sorted}(R, l_1) \wedge \mathsf{sorted}(R, l_2)$ $\wedge\ (\forall x\ y.\ \mathsf{mem}\ x\ l_1 \wedge \mathsf{mem}\ y\ l_2 \supset R\ x\ y)$ $\supset \mathsf{sorted}(R, l_1\ @\ l_2)$

Finally, we define what it means to be a sorting function parameterized by an order R.

Definition 51 (performs_sorting)

$$\mathsf{performs\_sorting}\ f\ R \equiv \forall l.\ \mathsf{perm}\ l\ (f(R,l)) \wedge \mathsf{sorted}(R, f(R,l)).$$

5.1.1 Naive Quicksort

We now define Quicksort and prove that it is a sorting function according to the above specifications.

$$\begin{array}{rcl} \mathsf{qsort}\ (ord, [\,]) & \equiv & [\,] \\ \mathsf{qsort}\ (ord, h :: t) & \equiv & \mathsf{qsort}\ (ord, \mathsf{filter}\ (\neg \circ ord\ h)\ t) \\ & & \quad @\ [h]\ @ \\ & & \mathsf{qsort}\ (ord, \mathsf{filter}\ (ord\ h)\ t). \end{array}$$

The termination relation for qsort is measure(length ∘ snd). TFL defines the function but the postprocessors are not able to prove the 2 termination conditions:

$$\begin{array}{l} \forall ord\ h\ t.\ \mathsf{length}\ (\mathsf{filter}\ (ord\ h)\ t) < \mathsf{length}(h :: t) \\ \forall ord\ h\ t.\ \mathsf{length}\ (\mathsf{filter}\ (\lambda h_1. \neg(ord\ h\ h_1))\ t) < \mathsf{length}(h :: t) \end{array}$$

These are however, quite simple to prove with the lemma

$$\mathsf{length}(\mathsf{filter}\ P\ L) \leq \mathsf{length}\ L.$$

After eliminating the termination conditions, the following induction theorem is available:

$$\begin{array}{l} \vdash \forall P.\ \ (\forall ord.\ P(ord, [\,])) \wedge \\ \qquad\quad (\forall ord\ x\ rst.\ P(ord, \mathsf{filter}(ord\ x)\ rst) \wedge \\ \qquad\qquad\qquad\qquad P\ (ord, \mathsf{filter}(\neg \circ ord\ x)\ rst)\ \supset P(ord, x :: rst)) \\ \qquad\quad \supset \forall v\ v_1.\ P(v, v_1). \end{array} \tag{5.1}$$

In order to prove correctness, the order relation supplied as an argument to Quicksort must be transitive and total.

Definition 52 (total)

$$\mathsf{total}(R) \equiv \forall x\ y.\ R\ x\ y \vee R\ y\ x.$$

The following lemmas then establish the correctness of Quicksort. All are straightforward to prove by induction with (5.1), using the lemmas already at hand. Note that *qsort_mem_stable* is only used to help prove *qsort_perm*.

qsort_mem_stable	$\mathsf{mem}\ x\ (\mathsf{qsort}(R, l)) = \mathsf{mem}\ x\ l$
qsort_perm	$\mathsf{perm}\ l\ (\mathsf{qsort}(R, l))$
qsort_orders	$\mathsf{transitive}\ R \wedge \mathsf{total}\ R \supset \mathsf{sorted}(R, \mathsf{qsort}(R, l))$
qsort_sorts	$\mathsf{transitive}\ R \wedge \mathsf{total}\ R \supset \mathsf{performs\_sorting}\ \mathsf{qsort}\ R$

5.1.2 Faster Quicksort

In qsort, the partitioning step traverses the list twice. Now we make a more reasonable implementation. To start, we define a function that partitions a list around a predicate, and builds two result lists as it goes.

$$\begin{array}{rcl} \mathsf{part}(P, [\,], l_1, l_2) & \equiv & (l_1, l_2) \\ \mathsf{part}(P, h :: t, l_1, l_2) & \equiv & \mathtt{if}\ P\ h\ \ \mathtt{then}\ \mathsf{part}(P, t, h :: l_1, l_2) \\ & & \qquad\quad\ \ \mathtt{else}\ \mathsf{part}(P, t, l_1, h :: l_2) \end{array}$$

A quicker Quicksort is then specified as follows. We neglect to give the termination relation:

$$\begin{array}{l} \mathsf{fqsort}(ord, [\,]) \equiv [\,] \\ \mathsf{fqsort}(ord, h :: t) \equiv \\ \qquad \mathtt{let}\ (l_1, l_2) = \mathsf{part}((\lambda y.\ ord\ y\ h, t), [\,], [\,]) \\ \qquad \mathtt{in} \\ \qquad\ \ \mathsf{fqsort}(ord, l_1)\ @\ [h]\ @\ \mathsf{fqsort}(ord, l_2). \end{array}$$

Examining the following theorem returned from the definition of fqsort, we see that 3 termination conditions have been placed on the assumptions. The conclusion is a conjunction. The first conjunct is the definition and the second is the principle of recursion induction for fqsort.

$$
\left[\begin{array}{l}
\mathsf{WF}\ R, \\
\forall l_1\ l_2\ ord\ h\ t.\ ((l_1,l_2) = \mathsf{part}((\boldsymbol{\lambda} y.\ ord\ y\ h), t, [\,], [\,])) \supset R(ord, l_2)(ord, h :: t), \\
\forall l_1\ l_2\ ord\ h\ t.\ ((l_1,l_2) = \mathsf{part}((\boldsymbol{\lambda} y.\ ord\ y\ h), t, [\,], [\,])) \supset R(ord, l_1)(ord, h :: t)
\end{array}\right]
$$

$$
\begin{array}{l}
\quad \vdash \\
((\mathsf{fqsort}(ord, [\,]) = [\,]) \wedge \\
\ (\mathsf{fqsort}(ord, h :: t) = \\
\qquad\quad \mathtt{let}\ (l_1, l_2) = \mathsf{part}((\boldsymbol{\lambda} y.\ ord\ y\ h), t, [\,], [\,]) \\
\qquad\quad \mathtt{in} \\
\qquad\quad \mathsf{fqsort}(ord, l_1)\ @\ [h]\ @\ \mathsf{fqsort}(ord, l_2))) \\
\ \wedge \\
\ (\forall P. \\
\quad (\forall ord.\ P(ord, [\,])) \wedge \\
\quad (\forall ord\ h\ t. \\
\qquad (\forall l_1 l_2.\ ((l_1, l_2) = \mathsf{part}((\boldsymbol{\lambda} y.\ ord\ y\ h), t, [\,], [\,])) \supset P(ord, l_2)) \wedge \\
\qquad (\forall l_1 l_2.\ ((l_1, l_2) = \mathsf{part}((\boldsymbol{\lambda} y.\ ord\ y\ h), t, [\,], [\,])) \supset P(ord, l_1)) \supset P(ord, h :: t)) \\
\quad \supset \forall v\ v_1.\ P(v, v_1))
\end{array}
$$

The user is now free to decide when termination is to be proved. In this example, we will ignore it, and merely focus on the proof of the permutation property:

$$\mathsf{perm}\ l\ (\mathsf{fqsort}(R, l)).$$

Proof. Start by applying the induction theorem. The base case is trivial, and the inductive case is the following goal (we do not show the termination conditions, which are also on the hypotheses):

$$
\begin{array}{l}
\mathsf{perm}(x :: rst) \\
\qquad (\mathtt{let}\ (l_1, l_2) = \mathsf{part}((\boldsymbol{\lambda} y.R\ y\ x), rst, [\,], [\,]) \\
\qquad\ \ \mathtt{in} \\
\qquad\ \ \mathsf{fqsort}(R, l_1) @ [x] @ \mathsf{fqsort}(R, l_2)) \\
\hline
3.\ \forall l_1 l_2.\ ((l_1, l_2) = \mathsf{part}((\boldsymbol{\lambda} y.Ryx), rst, [\,], [\,])) \supset \mathsf{perm}\ l_2\ (\mathsf{fqsort}(R, l_2)) \\
4.\ \forall l_1 l_2.\ ((l_1, l_2) = \mathsf{part}((\boldsymbol{\lambda} y.R\ y\ x), rst, [\,], [\,])) \supset \mathsf{perm}\ l_1\ (\mathsf{fqsort}(R, l_1))
\end{array}
$$

In order to use the inductive hypotheses, the entire `let` binding must somehow be brought to the top of the goal. Applying the following higher order rewrite rule

$$P(\mathtt{let}\ (x, y) = M\ \mathtt{in}\ N\ x\ y) = (\mathtt{let}\ (x, y) = M\ \mathtt{in}\ P(N\ x\ y)).$$

achieves this, giving the equivalent goal where the binding has been lifted to the top-level:

$$\begin{array}{l}
\quad \texttt{let } (x', y) = \mathsf{part}((\boldsymbol{\lambda} y.\ R\ y\ x), rst, [\,], [\,]) \\
\quad \texttt{in} \\
\quad \mathsf{perm}(x :: rst)\ (\mathsf{fqsort}(R, x')@[x]@\mathsf{fqsort}(R, y)) \\
\hline
3.\ \forall l_1 l_2.\ ((l_1, l_2) = \mathsf{part}((\boldsymbol{\lambda} y.Ryx), rst, [\,], [\,])) \supset \mathsf{perm}\ l_2\ (\mathsf{fqsort}(R, l_2)) \\
4.\ \forall l_1 l_2.\ ((l_1, l_2) = \mathsf{part}((\boldsymbol{\lambda} y.R\ y\ x), rst, [\,], [\,])) \supset \mathsf{perm}\ l_1\ (\mathsf{fqsort}(R, l_1))
\end{array}$$

Then it is straightforward to apply the following `let` introduction rule[1]

$$\frac{\Gamma, (vstruct = M) \vdash N}{\Gamma \vdash \texttt{let } vstruct = M \texttt{ in } N,}$$

which frees the binding $(x', y) = \mathsf{part}((\boldsymbol{\lambda} y.R\ y\ x), rst, [\,], [\,])$ and places it in the hypotheses.

$$\begin{array}{l}
\quad \mathsf{perm}(x :: rst)\ (\mathsf{fqsort}(R, x')@[x]@\mathsf{fqsort}(R, y)) \\
\hline
3.\ \forall l_1 l_2.\ ((l_1, l_2) = \mathsf{part}((\boldsymbol{\lambda} y.R\ y\ x), rst, [], [])) \supset \mathsf{perm}\ l_2\ (\mathsf{fqsort}(R, l_2)) \\
4.\ \forall l_1 l_2.\ ((l_1, l_2) = \mathsf{part}((\boldsymbol{\lambda} y.R\ y\ x), rst, [\,], [\,])) \supset \mathsf{perm}\ l_1\ (\mathsf{fqsort}(R, l_1)) \\
5.\ (x', y) = \mathsf{part}((\boldsymbol{\lambda} y.\ R\ y\ x), rst, [\,], [\,])
\end{array}$$

Now the special treatment given locally bound variables in the production of the induction theorem pays off, and forward chaining delivers the goal:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{perm}\ (x :: rst)\ (\mathsf{fqsort}(R, x')@[x]@\mathsf{fqsort}(R, y))} \\
\hline
5. & (x', y) = \mathsf{part}((\boldsymbol{\lambda} y.\ R\ y\ x), rst, [], []) \\
6. & \mathsf{perm}\ y\ (\mathsf{fqsort}(R, y)) \\
7. & \mathsf{perm}\ x'\ (\mathsf{fqsort}(R, x'))
\end{array}$$

Hypothesis 5 implies that $\mathsf{perm}\ rst\ (x'\ @\ y)$. Use of the lemmas *cons_perm*, *perm_trans*, and *perm_cong* finishes the proof.
□

One interesting aspect of this proof is the use of higher order rewriting in order to deal with the `let` construct. An alternative to the use of higher order rewriting would be to use first order rewriting with the definition of `let`. This approach has several drawbacks, among them being: the goal size can explode; the user's intuition is not well supported; and the inductive hypotheses must be subsequently be altered in order for them to be used, which may involve tedious revision in the hypotheses.

[1] A *vstruct* is an arbitrary tuple built up from (non-repeated) variables.

5.2 Iterated primitive recursion

The class of functions that can be proved to terminate with finite lexicographic combinations of the predecessor relation are known as the iterated primitive recursions. In this section, we examine a few of these, focusing mainly on how easy the termination relation is to express. The naive provers of the Appendix suffice to prove wellfoundedness and termination, which is to be expected, since iterated primitive recursions have a very regular syntax, which maps directly on to lexicographic combinations of the predecessor relation.

The first, and most famous is Ackermann's function. This celebrated example grows faster than any first order primitive recursive function. Its termination relation is $\mathsf{LEX\ pred\ pred}$:

$$\begin{array}{rcl}
\mathsf{ack}(\mathsf{0}, n) & = & n + 1 \\
\mathsf{ack}(\mathsf{Suc}\ m, \mathsf{0}) & \equiv & \mathsf{ack}(m, 1) \\
\mathsf{ack}(\mathsf{Suc}\ m, \mathsf{Suc}\ n) & \equiv & \mathsf{ack}(m, \mathsf{ack}(\mathsf{Suc}\ m, n))
\end{array}$$

The automatically extracted termination conditions

$$\begin{array}{l}
(\underline{\mathsf{Suc}\ m = \mathsf{Suc}\ m} \lor m = \mathsf{Suc}\ m \land \mathsf{0} = \mathsf{Suc}\ 1) \land \\
(\mathsf{Suc}\ m = \mathsf{Suc}(\mathsf{Suc}\ m) \lor \underline{\mathsf{Suc}\ m = \mathsf{Suc}\ m \land \mathsf{Suc}\ n = \mathsf{Suc}\ n}) \land \\
(\underline{\mathsf{Suc}\ m = \mathsf{Suc}\ m} \\
\quad \lor (m = \mathsf{Suc}\ m \land \\
\qquad (\mathsf{Suc}\ n = \mathsf{Suc}\ ((\mathsf{ack} \,|\, (\boldsymbol{\lambda}(s,t)(u,v).u = \mathsf{Suc}\ s \lor (s = u \land v = \mathsf{Suc}\ t)), \\
\qquad\qquad\qquad\qquad\qquad (\mathsf{Suc}\ m, \mathsf{Suc}\ n))(\mathsf{Suc}\ m, n))))
\end{array}$$

are trivial (as the underlined subterms show) and automatically proved by rewriting. The requested recursion equations are returned, along with the following induction theorem:

$$\begin{array}{l}
\forall P.\ (\forall n.\ P(\mathsf{0}, n)) \land \\
\quad (\forall m.\ P(m, 1) \supset P(\mathsf{Suc}\ m, \mathsf{0})) \land \\
\quad (\forall m\ n.\ P(m, \mathsf{ack}(\mathsf{Suc}\ m, n)) \land P(\mathsf{Suc}\ m, n) \supset P(\mathsf{Suc}\ m, \mathsf{Suc}\ n)) \\
\quad \supset \forall v\ v_1.\ P\ (v, v_1)
\end{array}$$

As a basic example, of the application of this theorem, a proof that the Ackermann function grows faster than the addition function is quite easy by use of the induction theorem:

$$\vdash \forall x\ y.\ x + y < \mathsf{ack}(x, y).$$

A closely associated function is *Sudan's* function:[2]

[2] According to a posting (Sept. 12, 1997) by Bill Dubuque on the `comp.theory` newsgroup, Sudan and Ackermann were both students of Hilbert. Sudan's function grows faster than

$$
\begin{array}{rcl}
\mathsf{Sudan}\ 0\ (x,y) & \equiv & x+y \\
\mathsf{Sudan}\ (\mathsf{Suc}\ n)(x,0) & \equiv & x \\
\mathsf{Sudan}\ (\mathsf{Suc}\ n)(x,\mathsf{Suc}\ y) & \equiv & \mathsf{Sudan}\ n\ \ (\mathsf{Sudan}\ (\mathsf{Suc}\ n)\ (x,y), \\
 & & \qquad\quad \mathsf{Sudan}\ (\mathsf{Suc}\ n)\ (x,y) + \mathsf{Suc}\ y)
\end{array}
$$

The termination relation for Sudan is LEX pred (inv_image pred snd). Again, the automated provers handle the termination and wellfoundedness of this function automatically.
Another iterated primitive recursion was part of a posting of the American logician Harvey Friedman on the 'FOM' (Foundations of Mathematics) mailing list on May 25, 1999.[3]

$$
\begin{array}{rcl}
\mathsf{V}\ (\mathsf{Suc}\ 0,n,m) & \equiv & n \\
\mathsf{V}\ (\mathsf{Suc}\ (\mathsf{Suc}\ k),n,\mathsf{Suc}\ 0) & \equiv & \mathsf{V}\ (\mathsf{Suc}\ k,\mathsf{Suc}\ n,\mathsf{Suc}\ n) \\
\mathsf{V}\ (\mathsf{Suc}\ (\mathsf{Suc}\ k),n,\mathsf{Suc}\ (\mathsf{Suc}\ m)) & \equiv & \mathsf{V}\ (\mathsf{Suc}\ k,\ \ \mathsf{V}\ (\mathsf{Suc}\ (\mathsf{Suc}\ k),n,\mathsf{Suc}\ m)+1, \\
 & & \qquad\quad \mathsf{V}\ (\mathsf{Suc}\ (\mathsf{Suc}\ k),n,\mathsf{Suc}\ m)+1)
\end{array}
$$

This function grows very fast; much faster than Ackermann's function. The termination relation for this function is LEX pred (LEX pred pred). Again, the automated provers handle the termination and wellfoundedness of this function automatically.

5.3 Propositional logic algorithms

In this section, we present the termination proofs of a couple of algorithms used to implement decision procedures for propositional logic. The first is due to Boyer and Moore, who implement a tautology checker by translating propositions to so-called 'IF'-trees, which are then reduced. The function and some variants, including a primitive recursive one, has been studied by Paulson [84]; the primitive recursive algorithm has also been extracted from a constructive proof of the specification in the COQ system. The second example is a rendition of what was probably the first implementation, by Hao Wang, of the sequent calculus [106]. After Wang's implementation, John McCarthy implemented the algorithm and

Ackermann's function (except at a single point), and was published first. However, Hilbert preferred Ackermann's function and the rest is history. Dubuque gives the reference (which I have not yet read)

> Calude, Cristian; Marcus, Solomon; Ţevy, Ionel The first example of a recursive function which is not primitive recursive. Historia Math. 6 (1979), no. 4, 380–384. MR 80i:03053 03D20 01A60

[3] Entitled *Mythical Trees.* The posting may be found at `http://www.math.psu.edu/simpson/fom/postings/9905.84`.

presented it in the Lisp 1.5 manual. In Cambridge undergraduate lecture notes, Martin Richards converted the Lisp presentation into ML, which is where the author first came across it.

5.3.1 Evaluation of conditional expressions

In this example, a logical datatype (cond) of conditional expressions with the following constructors is declared:

$$\begin{aligned} \mathsf{A} &: \mathsf{ind} \to \mathsf{cond} \\ \mathsf{IF} &: \mathsf{cond} \to \mathsf{cond} \to \mathsf{cond} \to \mathsf{cond} \end{aligned}$$

The following then defines a normalization function for such expressions.

$$\begin{aligned} \mathsf{norm}\ (\mathsf{A}\ i) &\equiv \mathsf{A}\ i \\ \mathsf{norm}\ (\mathsf{IF}(\mathsf{A}\ x)\ y\ z) &\equiv \mathsf{IF}(\mathsf{A}\ x)\ (\mathsf{norm}\ \ y)\ (\mathsf{norm}\ \ z) \\ \mathsf{norm}\ (\mathsf{IF}(\mathsf{IF}\ u\ v\ w)\ y\ z) &\equiv \mathsf{norm}\ (\mathsf{IF}\ u\ (\mathsf{IF}\ v\ y\ z)\ (\mathsf{IF}\ w\ y\ z)) \end{aligned}$$

The termination relation is measure M, where M, attributed to Robert Shostak, is defined by primitive recursion:

$$\begin{aligned} \mathsf{M}(\mathsf{A}\ i) &\equiv 1 \\ \mathsf{M}(\mathsf{IF}\ x\ y\ z) &\equiv \mathsf{M}x + (\mathsf{M}x * \mathsf{M}y) + (\mathsf{M}x * \mathsf{M}z) \end{aligned}$$

The system returns 3 termination conditions for norm :

$$\begin{aligned} \mathsf{M}z < \mathsf{M}(\mathsf{IF}(\mathsf{A}\ x)\ y\ z) &\quad \wedge \\ \mathsf{M}y < \mathsf{M}(\mathsf{IF}(\mathsf{A}\ x)\ y\ z) &\quad \wedge \\ \mathsf{M}(\mathsf{IF}\ u\ (\mathsf{IF}\ v\ y\ z)\ (\mathsf{IF}\ w\ y\ z)) &\quad < \quad \mathsf{M}(\mathsf{IF}(\mathsf{IF}\ u\ v\ w)\ y\ z) \end{aligned}$$

which when expanded with the definition of M, give the goal

$$\begin{aligned} &\mathsf{M}z < 1 + \mathsf{M}y + \mathsf{M}z \wedge \\ &\mathsf{M}y < 1 + \mathsf{M}y + \mathsf{M}z \wedge \\ &\mathsf{M}u + \mathsf{M}u * (\mathsf{M}v + \mathsf{M}v * \mathsf{M}y + \mathsf{M}v * \mathsf{M}z) + \\ &\mathsf{M}u * (\mathsf{M}w + \mathsf{M}w * \mathsf{M}y + \mathsf{M}w * \mathsf{M}z) \\ &< \\ &(\mathsf{M}u + \mathsf{M}u * \mathsf{M}v + \mathsf{M}u * \mathsf{M}w) + \\ &(\mathsf{M}u + \mathsf{M}u * \mathsf{M}v + \mathsf{M}u * \mathsf{M}w) * \mathsf{M}y + \\ &(\mathsf{M}u + \mathsf{M}u * \mathsf{M}v + \mathsf{M}u * \mathsf{M}w) * \mathsf{M}z \end{aligned}$$

The naive termination prover from the Appendix cannot prove this automatically, although the highly automated system of [39, 38] can. With our current toolset, however, the simple induction lemma $\vdash \forall x.\ 0 < \mathsf{M}x$ needs to be proved, and arithmetic laws for distribution of products over sums must also be manually applied to simplify the problem before the termination condition falls into the realm of linear arithmetic.

5.3.2 Wang's algorithm

Now we consider a propositional logic decision procedure first committed to machine by Hao Wang in 1958.

> *The whole program has about 1000 lines. The length of the sequents to be tested is deliberately confined to 72 symbols, so that each sequent can be presented by a single punched card. Although this restriction can be removed, it makes the coding considerably easier and gives ample room for handling the problems on hand.*
>
> – *Hao Wang*

Wang's algorithm implements the sequent calculus and provides a nice demonstration of pattern matching. Consider the type α prop of propositions, defined by the following datatype constructors:

$$\begin{array}{rcl}
\mathsf{VAR} & : & \alpha \rightarrow \alpha\ \mathsf{prop} \\
\mathsf{NOT} & : & \alpha\ \mathsf{prop} \rightarrow \alpha\ \mathsf{prop} \\
\mathsf{AND} & : & \alpha\ \mathsf{prop} \rightarrow \alpha\ \mathsf{prop} \rightarrow \alpha\ \mathsf{prop} \\
\mathsf{OR} & : & \alpha\ \mathsf{prop} \rightarrow \alpha\ \mathsf{prop} \rightarrow \alpha\ \mathsf{prop}
\end{array}$$

Wang's algorithm can be understood as a rewriting system on 4-tuples vl, l, r, vr representing sequents. In such a tuple, vl, l represents the left side of the sequent and r, vr represents the right. vl and vr are sets (here just lists) of variables, and l and r represent lists of compound formulae yet to be broken down. The algorithm repeatedly breaks leading compound formulae down. If a leading formula is a variable, it is shunted to the variable list. Eventually, no compound formulae remain: in the final step of the algorithm, both l and r are empty, and then the validity test merely involves checking whether vl and vr share a common element.

$$\begin{array}{lcl}
\mathsf{Prv}\ (vl, [\,], \mathsf{VAR}\ v :: r, vr) & \equiv & \mathsf{Prv}\ (vl, [\,], r, v :: vr) \\
\mathsf{Prv}\ (vl, [\,], \mathsf{NOT}\ x :: r, vr) & \equiv & \mathsf{Prv}\ (vl, [x], r, vr) \\
\mathsf{Prv}\ (vl, [\,], \mathsf{OR}\ x\ y :: r, vr) & \equiv & \mathsf{Prv}\ (vl, [\,], x :: y :: r, vr) \\
\mathsf{Prv}\ (vl, [\,], \mathsf{AND}\ x\ y :: r, vr) & \equiv & \mathsf{Prv}\ (vl, [\,], x :: r, vr) \wedge \mathsf{Prv}\ (vl, [\,], y :: r, vr) \\
 & & \\
\mathsf{Prv}\ (vl, \mathsf{VAR}\ v :: l, r, vr) & \equiv & \mathsf{Prv}\ (v :: vl, l, r, vr) \\
\mathsf{Prv}\ (vl, \mathsf{NOT}\ x :: l, r, vr) & \equiv & \mathsf{Prv}\ (vl, l, x :: r, vr) \\
\mathsf{Prv}\ (vl, \mathsf{AND}\ x\ y :: l, r, vr) & \equiv & \mathsf{Prv}\ (vl, x :: y :: l, r, vr) \\
\mathsf{Prv}\ (vl, \mathsf{OR}\ x\ y :: l, r, vr) & \equiv & \mathsf{Prv}\ (vl, x :: l, r, vr) \wedge \mathsf{Prv}\ (vl, y :: l, r, vr) \\
 & & \\
\mathsf{Prv}\ (vl, [\,], [\,], vr) & \equiv & \exists y.\ \mathsf{mem}\ y\ vl \wedge \mathsf{mem}\ y\ vr
\end{array}$$

The top-level invocation of the procedure is as follows:

$$\mathsf{Prove}\ P \equiv \mathsf{Prv}\ ([\,],[\,],[P],[\,]).$$

The termination of Prv can be shown with the help of a non-standard size measure much like that for norm. What is required is to make a two-argument proposition bigger than a list of two propositions:

$$\begin{array}{rcl} \mathsf{Meas}\ (\mathsf{VAR}\ v) & \equiv & 0 \\ \mathsf{Meas}\ (\mathsf{NOT}\ x) & \equiv & \mathsf{Suc}\ (\mathsf{Meas}\ x) \\ \mathsf{Meas}\ (\mathsf{AND}\ x\ y) & \equiv & \mathsf{Meas}\ x + \mathsf{Meas}\ y + 2 \\ \mathsf{Meas}\ (\mathsf{OR}\ x\ y) & \equiv & \mathsf{Meas}\ x + \mathsf{Meas}\ y + 2 \end{array}$$

The parameterized size function for lists helps express the termination relation (the sum of the sizes of l and r decreases):

$$\mathsf{measure}(\boldsymbol{\lambda}(w,x,y,z).\ \mathsf{list\_size\ Meas}\ x + \mathsf{list\_size\ Meas}\ y).$$

The termination conditions are then straightforward to prove. However, noticing that the definition is nearly tail-recursive—always a good excuse to use the multiset extension—an alternative and in some sense simpler termination relation is the following:

$$\begin{array}{l} \mathsf{inv\_image}\ (\mathsf{predMset}\ (\mathsf{measure}\ (\mathsf{prop\_size}\ (\boldsymbol{\lambda} v.0)))) \\ \qquad\qquad (\boldsymbol{\lambda}(w,x,y,z).\ \mathsf{list\_to\_mset}\ (x\ @\ y)). \end{array}$$

That is, the multiset formed by the second and third arguments decreases in each recursive call. The advantage of using this termination relation is that no special notion of size has to be invented: the following 'standard' size definition (discussed in Section 2.6.1) for α prop suffices (indeed, a definition that just counts the number of constructors would also work), therefore the amount of intelligence needed to find a correct termination relation is much less than the previous solution.

$$\begin{array}{rcl} \mathsf{prop\_size}\ f\ (\mathsf{VAR}\ x) & \equiv & 1 + fx \\ \mathsf{prop\_size}\ f\ (\mathsf{NOT}\ p) & \equiv & 1 + \mathsf{prop\_size}\ f\ p \\ \mathsf{prop\_size}\ f\ (\mathsf{OR}\ p1\ p2) & \equiv & 1 + \mathsf{prop\_size}\ f\ p1 + \mathsf{prop\_size}\ f\ p2 \\ \mathsf{prop\_size}\ f\ (\mathsf{AND}\ p1\ p2) & \equiv & 1 + \mathsf{prop\_size}\ f\ p1 + \mathsf{prop\_size}\ f\ p2. \end{array}$$

5.4 $f(x) = f(x+1)$

How does TFL handle the case of a patently non-terminating function specification when a relationless definition is made? The result of attempting the proposed definition is (ignoring the induction theorem):

$$\left[\begin{array}{l}\mathsf{WF}\ R,\\ \forall x.\ R\ (x+1)\ x\end{array}\right] \vdash f(x) = f(x+1). \tag{5.2}$$

An alternate characterization of wellfoundedness is the absence of infinite decreasing chains:

$$\forall R.\ \mathsf{WF}\ R = \neg(\exists f.\ \forall n.\ R\ (f\ (\mathsf{Suc}\ n))\ (f\ n)).$$

Proof ($\Rightarrow$) By contraposition, suppose there is an infinite decreasing chain. This describes a set with no R-minimal element.

($\Leftarrow$) By contraposition, suppose there is a set with no R-minimal element. The desired function is then obtained by a direct application of the axiom of Dependent Choice:

$$\begin{array}{l} P\ a \wedge (\forall x.\ P\ x \supset \exists y.\ P\ y \wedge R\ x\ y)\\ \quad\supset\\ \exists f.\ (f\ \mathsf{0} = a) \wedge \forall n.\ P\ (f\ n) \wedge R\ (f\ n)\ (f\ (\mathsf{Suc}\ n)) \end{array}$$

□

With this variant of wellfoundedness, it is easy to show that the termination conditions are unsatisfiable, and thus that (5.2) reduces to $\vdash$ True.

Of course, not all non-terminating recursions are as easy to detect as in this example; however, our deductive approach ensures that the termination constraints will propagate faithfully through inference steps, thus preserving soundness, no matter what kind of foolish input has been given.

5.5 Higher order recursion

A so-called *higher order* recursion is a one in which a higher order function is employed to apply the function under definition to its arguments. For example, consider a datatype of labelled, finitely branching trees defined by the constructor

$$\boxed{\mathsf{Node} : \alpha \to \alpha\ \mathsf{tree\ list} \to \alpha\ \mathsf{tree}}$$

An 'occurs check' for labels in this type can be written as follows:

$$\mathsf{occurs}\ (x, \mathsf{Node}\ v\ tl) \equiv (x = v) \vee \mathsf{exists}\ (\boldsymbol{\lambda} t.\ \mathsf{occurs}\ (x,t))\ tl.$$

If this recursion equation is processed in a standard notion of context, *i.e.*, one holding only congruence rules for datatypes, `if _ then _ else _`, and `let`, the extraction process of Section 3.2 will yield the termination conditions

$$\exists R.\ \mathsf{WF}\ R \wedge R\ (x,\ t)\ (x,\ \mathsf{Node}\ v\ tl),$$

which are unprovable because there is no relationship between t and $\mathsf{Node}\ v\ tl$. The missing insight is that each tree in the list tl will be a proper subterm of $\mathsf{Node}\ v\ tl$. Somewhat surprisingly, this insight can be captured with the following congruence rule:

$$\begin{array}{l} (\ell_1 = \ell_2) \wedge (\forall y.\ \mathsf{mem}\ y\ \ell_2 \supset (P_1\ y = P_2\ y)) \\ \quad \supset \\ \mathsf{exists}\ P_1\ \ell_1 = \mathsf{exists}\ P_2\ \ell_2. \end{array} \tag{5.3}$$

After manually adding (5.3) to the congruences known to the extraction mechanism described in Section 3.2, trying the definition again yields a more satisfactory result:

$$\begin{array}{l} \left[\begin{array}{l} \mathsf{WF}\ R, \\ \forall t\ tl\ x\ v.\ \mathsf{mem}\ t\ tl \supset R\ (x,\ t)\ (x,\ \mathsf{Node}\ v\ tl) \end{array}\right] \\ \quad \vdash \\ (\mathsf{occurs}\ (x, \mathsf{Node}\ v\ tl) = (x = v) \vee \mathsf{exists}\ (\lambda t.\ \mathsf{occurs}\ (x,t))\ tl) \\ \wedge \\ \forall P.\ (\forall x\ v\ tl.\ (\forall t.\ \mathsf{mem}\ t\ tl \supset P\ (x,t)) \supset P\ (x, \mathsf{Node}\ v\ tl)) \supset \forall v\ v_1.\ P\ (v, v_1) \end{array}$$

Now termination is provable, *e.g.*, by setting $R \mapsto \mathsf{measure}\ (\mathsf{tree\_size} \circ \mathsf{snd})$ (where $\mathsf{tree\_size}$ counts the number of constructors in the tree).

Thus it seems that higher order functions need to have congruence theorems proved for them, in order to facilitate subsequent definitions by higher order recursion. The following are suitable congruence rules for two other standard higher order functions over lists:

$$(\ell_1 = \ell_2) \wedge (\forall y.\ \mathsf{mem}\ y\ \ell_2 \supset (f_1\ y = f_2\ y) \supset \mathsf{map}\ f_1\ \ell_1 = \mathsf{map}\ f_2\ \ell_2$$

$$\begin{array}{l} (\ell_1 = \ell_2) \wedge (b_1 = b_2) \wedge (\forall x\ a.\ \mathsf{mem}\ x\ \ell_2 \supset (f_1\ x\ a = f_2\ x\ a)) \\ \quad \supset \\ \mathsf{rev\_itlist}\ f_1\ \ell_1\ b_1 = \mathsf{rev\_itlist}\ f_2\ \ell_2\ b_2 \end{array}$$

Using the latter, we can define a parameterized notion of size for tree of the kind discussed in Section 2.6.1. The input

$$\mathsf{tree\_size}\ f\ (\mathsf{Node}\ v\ tl) \equiv \mathsf{rev\_itlist}\ (\lambda h\ i.\ \mathsf{tree\_size}\ f\ h\ + i)\ tl\ (1 + f\ v).$$

yields

$$\left[\begin{array}{l}\mathsf{WF}\ R,\\ \forall v\ tl\ f\ h.\ \mathsf{mem}\ h\ tl \supset R\ (f,\ h)\ (f,\ \mathsf{Node}\ v\ tl)\end{array}\right]$$

$$\vdash$$

$$\begin{array}{l}(\mathsf{tree\_size}\ f\ (\mathsf{Node}\ v\ tl) \equiv \mathsf{rev\_itlist}\ (\lambda h\ i.\ \mathsf{tree\_size}\ f\ h\ + i)\ tl\ (1 + f\ v))\\ \wedge\\ (\forall P.\ (\forall f\ v\ tl.\ (\forall h.\ \mathsf{mem}\ h\ tl \supset P\ f\ h) \supset P\ f\ (\mathsf{Node}\ v\ tl)) \supset \forall v\ v_1.\ P\ v\ v_1).\end{array}$$

Again termination is provable by setting $R \mapsto \mathsf{measure}\ (\mathsf{tree\_size} \circ \mathsf{snd})$.

Although this manual use of congruence rules enables the proper definition of higher order recursions, it has the current defect that rules have to be installed by hand. It may be possible to automate the proof of standard congruence rules for higher order functions, where 'standard' means using the idea of proper subterm (mem in the case of the list type) to build the target congruence rule. However, this raises the problem that there may be higher order recursions that need a stronger notion of context in order for provable termination conditions to be extracted for them. It may be for such reasons that the authors of [36] couldn't find a general way to handle higher order recursion. In any case, if a stronger notion of context is needed, then the design of TFL allows the user to prove the required theorem and install it in the database of congruence rules. Thus one of the advantages of our deductive approach is its flexibility in comparison to the meta-theoretical tack.

The ability to define functions by higher order recursion over 'nested' datatypes such as tree might seem to toll the death knell for the function definition facilities in nested datatype packages of the sort written by Gunter, Harrison, and Berghofer and Wenzel [46, 48, 11]. To a large extent, that is true. However, in order to prove totality of higher order recursions, one typically uses a 'size' measure (as above), and—as a bootstrapping step—that would currently be most easily defined with the function definition facilities provided by the datatype package.

5.6 Program transformations

A *scheme* is (loosely) defined to be a set of recursion equations with some free variables occurring in the right hand side, but not the left. For example, in the scheme

$$\mathsf{linRec}(x) \equiv \mathtt{if}\ \mathit{atomic}\ x\ \mathtt{then}\ A\ x\ \mathtt{else}\ \mathit{join}\ (\mathsf{linRec}\ (\mathit{dest}\ x))\ (D\ x),$$

the variables *atomic*, *join*, *dest*, *A*, and *D* do not occur as arguments to linRec. On the face of it, this is not a definition, but, as was shown in Section 3.6, a constrained definition can be built from this description. One reason why schemes are interesting is that high-level equivalences between classes of programs can be expressed with them. Thus schemes serve as a foundation for the field of *program transformation*, which is an approach to program development in which a high-level algorithm, implementing a specification, serves as the starting point for a sequence of formal transformation steps intended at improving the program in some way (usually with respect to time or space efficiency). The book [81] contains a wealth of examples summarizing the state of the art as of the late 1980s.

The linRec scheme expresses a class of *linear recursive* programs. Under certain conditions, instances of linRec are equal to corresponding instances of the following tail-recursive scheme, which uses an accumulating parameter:

$$\begin{array}{rl} \mathsf{accRec}(x,u) \equiv & \texttt{if}\ \mathit{atomic}\ x\ \texttt{then}\ \mathit{join}\ (A\ x)\ u \\ & \texttt{else}\ \mathsf{accRec}\ (\mathit{dest}\ x,\ \mathit{join}\ (D\ x)\ u). \end{array}$$

Intuitively, the recursive calls of linRec must get 'stacked up' somehow,[4] waiting for deeper recursive calls to return. In contrast, calls to accRec need not be stacked, even mentally. If the combination function *join* is associative, then the implicit bracketing of the stacked recursive calls can be replaced with a single data value that gets passed and modified at each recursive call.

We now formalize this intuition. The result of the TFL definition of linRec is:

$$\begin{array}{l} \left[\begin{array}{l} \mathsf{WF}\ R, \\ \forall x.\ \neg(\mathit{atomic}\ x) \supset R\ (\mathit{dest}\ x)\ x \end{array}\right] \\ \quad \vdash \\ (\mathsf{linRec}\ D\ \mathit{dest}\ \mathit{join}\ A\ \mathit{atomic}\ x = \\ \quad \texttt{if}\ \mathit{atomic}\ x\ \texttt{then}\ A\ x \\ \quad \texttt{else}\ \mathit{join}\ (\mathsf{linRec}\ D\ \mathit{dest}\ \mathit{join}\ A\ \mathit{atomic}\ (\mathit{dest}\ x))\ (D\ x)) \\ \wedge \\ \forall P.\ (\forall x.\ (\neg(\mathit{atomic}\ x) \supset P\ (\mathit{dest}\ x)) \supset P\ x) \supset \forall v.\ P\ v, \end{array}$$

and that for accRec is

$$\begin{array}{l} \left[\begin{array}{l} \mathsf{WF}\ R, \\ \forall x.\ \neg(\mathit{atomic}\ x) \supset R\ (\mathit{dest}\ x)\ x \end{array}\right] \\ \quad \vdash \\ (\mathsf{accRec}\ D\ \mathit{dest}\ A\ \mathit{join}\ \mathit{atomic}\ (x,u) = \\ \quad \texttt{if}\ \mathit{atomic}\ x\ \texttt{then}\ \mathit{join}\ (A\ x)\ u \\ \quad \texttt{else}\ \mathit{accRec}\ D\ \mathit{dest}\ A\ \mathit{join}\ \mathit{atomic}\ (\mathit{dest}\ x, \mathit{join}\ (D\ x)\ u)) \\ \wedge \\ \forall P.\ (\forall x\ u.\ (\neg(\mathit{atomic}\ x) \supset P\ (\mathit{dest}\ x, \mathit{join}\ (D\ x)\ u)) \supset P\ (x,u)) \\ \quad \supset \forall v\ v_1.\ P\ (v,v_1). \end{array}$$

The formal program transformation is then captured in the following theorem:

[4] Of course, this is just a mental picture: there is no need to expect a runtime system to use the stack; *e.g.*, some ML compilers allocate function calls in the heap.

$$\left[\begin{array}{l}\mathsf{WF}\ R,\\ (\forall x.\ \neg(\mathit{atomic}\ x) \supset R\ (\mathit{dest}\ x)\ x),\\ (\forall p\ q\ r.\ \mathit{join}\ p\ (\mathit{join}\ q\ r) = \mathit{join}\ (\mathit{join}\ p\ q)\ r)\end{array}\right] \\ \vdash \\ \forall x\ u.\ \mathit{join}\ (\mathsf{linRec}\ D\ \mathit{dest}\ \mathit{join}\ A\ \mathit{atomic}\ x)\ u \\ = \\ \mathsf{accRec}\ D\ \mathit{dest}\ A\ \mathit{join}\ \mathit{atomic}\ (x, u). \tag{5.4}$$

Proof. Apply the induction theorem for accRec, then expand the definitions of linRec and accRec.
□

Example. The factorial function supplies a simple instantiation of this theorem:

$$\mathsf{fact}(x) \equiv \texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * \mathsf{fact}(x-1).$$

By inspection, fact can be seen to be equivalent to instantiating linRec with the following substitutions:

$$\begin{array}{rcl} \mathit{atomic} & \mapsto & \boldsymbol{\lambda} x.\ x = 0 \\ A & \mapsto & \boldsymbol{\lambda} x.\ 1 \\ \mathit{dest} & \mapsto & \boldsymbol{\lambda} x. x - 1 \\ \mathit{join} & \mapsto & \boldsymbol{\lambda} x\ y. x * y \\ D & \mapsto & \boldsymbol{\lambda} x. x \end{array}$$

Similarly, applying these substitutions to accRec (and abbreviating the result by the constant tailfact), yields

$$\begin{array}{rl} \mathsf{tailfact}(x, u) \equiv & \texttt{if}\ x = 0\ \texttt{then}\ u \\ & \texttt{else}\ \mathsf{tailfact}\ (x - 1,\ x * u). \end{array}$$

Now the initial value for the accumulator $u \mapsto 1$ must be invented, and then the instantiated (5.4) is

$$\left[\begin{array}{l}\mathsf{WF}\ R,\\ (\forall x.\ \neg(x = 0) \supset R\ (x-1)\ x),\\ (\forall p\ q\ r.\ p * q * r = (p * q) * r\end{array}\right] \vdash \forall x.\ \mathsf{fact}(x) = \mathsf{tailfact}(x, 1).$$

The assumptions are easy to eliminate, and finally the desired equivalence is obtained. This example reveals some of the difficulty in mechanizing program transformation: although the transformations are often easy to derive, the systematic application of them brings up difficulties at various stages, *e.g.*, the 'by inspection' steps, and also the need to invent values such as u above.
□

5.6.1 Unfold

This is an example originally presented by Bird [12], and later mechanized by Shankar [95]. Consider a datatype btree of binary trees with constructors

$$\begin{aligned}\mathsf{LEAF} &: \alpha\ \mathsf{btree}\\ \mathsf{NODE} &: \alpha\ \mathsf{btree} \to \alpha \to \alpha\ \mathsf{btree} \to \alpha\ \mathsf{btree}.\end{aligned}$$

The so-called *catamorphism* (primitive recursor) for this type is

$$\begin{aligned}\mathsf{btreeRec}\ \mathsf{LEAF}\ v\ f &\equiv v\\ \mathsf{btreeRec}\ (\mathsf{NODE}\ t_1\ M\ t_2)\ v\ f &\equiv f\ (\mathsf{btreeRec}\ t_1\ v\ f)\ M\ (\mathsf{btreeRec}\ t_2\ v\ f).\end{aligned}$$

This sort of structural recursion is straightforward to define; most higher order logic systems automate such definitions (see Section 2.2). However, the so-called *anamorphism* (or unfold, or co-recursor) for this type has not, until now, been straightforward to define in these systems. Understanding the following definition of unfold : $\alpha \to \beta$ btree may be eased by considering it as operating over an abstract datatype α which supports operations $more : \alpha \to$ bool and $dest : \alpha \to \alpha * \beta * \alpha$.

```
unfold $x$ ≡ if $more\ x$
             then let $(y_1, b, y_2) = dest\ x$
                  in
                    NODE (unfold $y_1$) $b$ (unfold $y_2$)
             else LEAF.
```

The automatically computed constraints attached to the definition are the following (the system is not currently smart enough to know that the two termination conditions share the same context):

$$\begin{bmatrix}\mathsf{WF}\ R,\\ \forall x\ y_1\ b\ y_2.\ more\ x \wedge ((y_1, b, y_2) = dest\ x) \supset R\ y_2\ x,\\ \forall x\ y_1\ b\ y_2.\ more\ x \wedge ((y_1, b, y_2) = dest\ x) \supset R\ y_1\ x.\end{bmatrix}$$

After some trivial manipulation to join the two termination conditions, the induction theorem for unfold is the following (omitting the hypotheses):

$$\begin{aligned}\forall P.\ (\forall x.\ (\forall y_1\ b\ y_2. more\ x \wedge ((y_1, b, y_2) = dest\ x) \supset P\ y_1 \wedge P\ y_2) \supset P\ x)\\ \supset \forall v.\ P\ v.\end{aligned} \tag{5.5}$$

It is easy to generalize unfold to an arbitrary range type by replacing NODE and LEAF with parameters g and c:

```
fuse $x$ ≡ if $more\ x$
           then let $(y_1, b, y_2) = dest\ x$
                in
                  $g$ (fuse $y_1$) $b$ (fuse $y_2$)
           else $c$.
```

The *fusion* theorem states that unfolding into a btree and then applying a structural recursive function to the result is equivalent to interweaving unfolding steps with the steps taken in the structural recursion. Thus two recursive passes over the data can be replaced by one:

$$\begin{array}{l}
\left[\begin{array}{l}\mathsf{WF}\ R,\\ \forall x\ y_1\ b\ y_2.\ \mathit{more}\ x \wedge ((y_1, b, y2) = \mathit{dest}\ x) \supset R\ y_1\ x\ \wedge R\ y_2\ x\end{array}\right]\\
\quad\vdash\\
\forall x\ c\ g.\ \mathsf{btreeRec}\ (\mathsf{unfold}\ \mathit{dest}\ \mathit{more}\ x)\ c\ g = \mathsf{fuse}\ c\ \mathit{dest}\ g\ \mathit{more}\ x.
\end{array}$$

Proof. The proof is by induction using (5.5), followed by expanding the definitions of btreeRec, unfold, and fuse.
□

5.6.2 Binary recursion

Now we look at a divide-and-conquer scheme that splits the input in two and recurses on the results. The binary recursion scheme splits its argument in two by applying the parameters *left* and *right*.

$$\begin{array}{ll}
\mathsf{binRec}(x) \equiv & \mathtt{if}\ \mathit{atomic}\ x\ \mathtt{then}\ A\ x\\
& \mathtt{else}\ \mathit{join}\ (\mathsf{binRec}\ (\mathit{left}\ x))\ (\mathsf{binRec}\ (\mathit{right}\ x))
\end{array}$$

The result of our definition is

$$\begin{array}{l}
\left[\begin{array}{l}\mathsf{WF}\ R,\\ \forall x.\ \neg(\mathit{atomic}\ x) \supset R\ (\mathit{right}\ x)\ x,\\ \forall x.\ \neg(\mathit{atomic}\ x) \supset R\ (\mathit{left}\ x)\ x\end{array}\right]\\
\quad\vdash\\
(\mathsf{binRec}\ \ \mathit{right}\ \mathit{left}\ \mathit{join}\ A\ \mathit{atomic}\ x =\\
\qquad \mathtt{if}\ \mathit{atomic}\ x\ \mathtt{then}\ A\ x\\
\qquad \mathtt{else}\ \mathit{join}\ (\mathsf{binRec}\ \ \mathit{right}\ \mathit{left}\ \mathit{join}\ A\ \mathit{atomic}\ (\mathit{left}\ x))\\
\qquad\qquad\qquad (\mathsf{binRec}\ \ \mathit{right}\ \mathit{left}\ \mathit{join}\ A\ \mathit{atomic}\ (\mathit{right}\ x)))\\
\wedge\\
\forall P.\ (\forall x.\ (\neg(\mathit{atomic}\ x) \supset P(\mathit{right}\ x)) \wedge\\
\qquad\quad (\neg(\mathit{atomic}\ x) \supset P(\mathit{left}\ x)) \supset P x)\\
\quad \supset \forall v.\ P\ v.
\end{array}$$

The example comes from Wand [105], who used paper and pencil, and has recently been treated by Shankar, using PVS [95]. In his original development, Wand was interested in explaining how continuations give the programmer a representation of the runtime stack, and thus can act as a bridge in the transformation of non-tail-recursive functions to tail recursive ones. In our development, we will avoid the continuation passing step (although it is simple for us to handle) and transform directly to tail recursion.

Now a general tail recursion scheme for lists is defined. In the definition, the parameter $\mathit{dest} : \alpha \to \alpha\ \mathsf{list}$ breaks the head h of the work list $h :: t$ into a list of new work, which it prepends to t before continuing. Our definition is quite general because the argument to the tail call may increase in length by any finite amount. (Wand and Shankar only consider tail recursions in which the *dest* parameter can produce two new pieces of work.)

$$\begin{array}{rcl} \mathsf{baRec}([\,],v) & \equiv & v \\ \mathsf{baRec}(h :: t, v) & \equiv & \texttt{if}\ \mathit{atomic}\ h\ \texttt{then}\ \mathsf{baRec}\ (t, \mathit{join}\ v\ (A\ h)) \\ & & \texttt{else}\ \mathsf{baRec}\ (\mathit{dest}\ h\ @\ t,\ v) \end{array}$$

The result of this definition is

$$\begin{array}{l}
\left[\begin{array}{l} \mathsf{WF}\ R, \\ \forall v\ t\ h.\ \neg(\mathit{atomic}\ h) \supset R\ (\mathit{dest}\ h\ @\ t,\ v)\ (h :: t, v), \\ \forall v\ t\ h.\ \mathit{atomic}\ h \supset R\ (t, \mathit{join}\ v\ (A\ h))\ (h :: t, v) \end{array}\right] \\
\quad \vdash \\
(\mathsf{baRec}\ \mathit{dest}\ A\ \mathit{join}\ \mathit{atomic}\ ([\,],v) = v) \wedge \\
(\mathsf{baRec}\ \mathit{dest}\ A\ \mathit{join}\ \mathit{atomic}\ (h :: t, v) = \\
\qquad \texttt{if}\ \mathit{atomic}\ h \\
\qquad \texttt{then}\ \mathsf{baRec}\ \mathit{dest}\ A\ \mathit{join}\ \mathit{atomic}\ (t, \mathit{join}\ v\ (A\ h)) \\
\qquad \texttt{else}\ \mathsf{baRec}\ \mathit{dest}\ A\ \mathit{join}\ \mathit{atomic}\ (\mathit{dest}\ h\ @\ t, v)) \\
\wedge \\
\forall P.\ (\forall v.\ P\ ([\,],v)) \wedge \\
\qquad (\forall h\ t\ v.\ \neg(\mathit{atomic}\ h) \supset P\ (\mathit{dest}\ h\ @\ t, v) \wedge \\
\qquad\qquad\qquad \mathit{atomic}\ h \supset P\ (t, \mathit{join}\ v\ (A\ h)) \supset P\ (h :: t, v)) \\
\supset \\
\qquad \forall v\ v_1.\ P\ (v, v_1)
\end{array}$$

We intend to prove an equivalence between binRec and baRec, but if we are not careful, the transformation will require the constraints for both binRec and baRec to be satisfied. However, a bit of thought reveals that the multiset extension allows one of the constraints to be expressed in terms of the other. In particular, the termination condition of baRec can be reduced to the (simpler) one of binRec:

$$\begin{array}{l}
\mathsf{WF}\ R \wedge (\forall h\ y.\ \neg atomic\ h \wedge \mathsf{mem}\ y\ (dest\ h) \supset R\ y\ h) \\
\quad\supset \\
\exists R'.\ \ \mathsf{WF}\ R' \wedge \\
\qquad (\forall h\ t\ v.\ \neg atomic\ h \supset R'\ (dest\ h\ @\ t,\ v)\ (h :: t, v)) \wedge \\
\qquad (\forall h\ t\ v.\ \ atomic\ h \supset R'\ (t,\ join\ v\ (A\ h))\ (h :: t, v))
\end{array}$$

Proof. Instantiate R' to inv_image (predMset R) (list_to_mset ∘ fst). This is well-founded by Theorems 33 and 35. The next conjunct is true by the assumptions and Definition 34, and the final conjunct is also true by Definition 34, since no elements are being put back into the multiset.

□

With this reduction, we can state and prove the following general theorem relating binary recursion and tail recursion. The essential insight is that the work list l of baRec represents a linearization of the binary tree of calls of binRec. Thus going from left to right through the work list, invoking binRec and accumulating the results, should deliver the same answer as executing baRec on the work list.

$$\begin{array}{l}
\left[\begin{array}{l}
\mathsf{WF}\ R, \\
\forall x.\ \neg atomic\ x \supset R\ (left\ x)\ x \wedge R\ (right\ x)\ x, \\
\forall p\ q\ r.\ join\ (join\ p\ q)\ r = join\ p\ (join\ q\ r)
\end{array}\right] \\
\quad \vdash \\
\forall l\ v_0. \\
\quad \mathsf{rev\_itlist}(\boldsymbol{\lambda} tr\ v.\ join\ v\ (\mathsf{binRec}\ right\ left\ join\ A\ atomic\ tr))\ l\ v_0 \\
\qquad\qquad = \\
\quad \mathsf{baRec}(\boldsymbol{\lambda} x.[left\ x,\ right\ x])\ A\ join\ atomic\ (l, v_0)
\end{array}$$

Proof. Induct with the induction theorem for baRec. The base case is straightforward; the step case is also essentially trivial, since it only involves using the induction hypotheses and rewriting with the definitions of rev_itlist, baRec, and binRec.

□

Finally, the specific equivalence desired can be obtained by instantiating the work list l to comprise the initial item of work $[x]$, and then reducing the definition of rev_itlist away.

$$\begin{array}{l}
\left[\begin{array}{l}
\mathsf{WF}\ R, \\
\forall x.\ \neg \mathit{atomic}\ x \supset R\ (\mathit{left}\ x)\ x \wedge R\ (\mathit{right}\ x)\ x, \\
\forall p\ q\ r.\ \mathit{join}\ (\mathit{join}\ p\ q)\ r = \mathit{join}\ p\ (\mathit{join}\ q\ r)
\end{array}\right] \\
\ \vdash \\
\forall x\ v_0. \\
\qquad \mathit{join}\ v_0\ (\mathsf{binRec}\ \mathit{right}\ \mathit{left}\ \mathit{join}\ A\ \mathit{atomic}\ x) \\
\qquad\quad = \\
\qquad \mathsf{baRec}\ (\boldsymbol{\lambda} x.\ [\mathit{left}\ x,\ \mathit{right}\ x])\ A\ \mathit{join}\ \mathit{atomic}\ ([x], v_0)
\end{array}$$

5.6.3 Related work

The paper by Huet and Lang [53] is an important early milestone in the field of program transformation. They worked in the LCF system, using fixpoint induction to derive program transformations. Program schemes were not defined; instead, transformations were represented via applications of the Y combinator, *i.e.*, had the form *applicability conditions* $\supset \mathsf{Y}\ \mathcal{F} = \mathsf{Y}\ \mathcal{G}$, for functionals $\mathcal{F}$ and $\mathcal{G}$. An influential aspect of the work was the use of second order matching to automate the application of program transformations.

The paper of Basin and Anderson [5] has much in common with our work: for example, both approaches represent schemes and transformations by HOL theorems (Basin and Anderson call these *rules*). Their work differs from ours by focusing on relations (they are interested in modelling logic programs) rather than functions. They present two techniques: in the first, program schemes are not defined; instead, transformations are derived by wellfounded induction on the arguments of the specified recursive relations (the relations themselves are left as variables). In the second, a program scheme is represented by an inductively defined relation. The first approach suffers from lack of automation: termination constraints are not synthesized and induction theorems are not automatically derived. In contrast, their second approach requires no mention of wellfoundedness, and induction is automatically derived by the inductive definition package of Isabelle/HOL.

Work using PVS has represented program schemes and transformations by theories parameterized over the parameters of the scheme and having as proof obligations the applicability conditions of the transformation [95, 33, 34]. To apply the program transformation, the theory must be instantiated, and the corresponding concrete proof obligations proved.

In our technique—in contrast—the parameters of a scheme are arguments to the defined constant, and the proof obligations are constraints on the recursion equations and the induction theorem. Thus, theorems are used to represent both program schemas and program transformations. Instantiating a program transformation in our setting merely requires one to instantiate type variables and/or free term variables in a theorem. It remains to be seen if one representation is preferable to the other. In other ways, however, our approach seems to offer improved functionality:

1. Currently, our technique produces more general schemes, since termination conditions are phrased in terms of an arbitrary wellfounded relation, whereas termination relations in PVS are restricted to measure functions [79]. Similarly, a general induction theorem is automatically derived for each scheme in our setting, whereas the PVS user is limited to measure induction (or may alternatively derive a more general induction theorem 'by hand' from wellfounded induction).

2. Our technique is more convenient because it automatically generates—by deductive steps—termination conditions for schemes. Taking the example of unfold, one doesn't have to ponder the right constraints in our setting: they are delivered as part of the returned definition. In contrast, the definition of unfold in [95] requires expert knowledge of the PVS type system in order to phrase the right constraints on the *Dest* parameter. Since the termination conditions of a scheme constrain any program transformation that mentions the scheme, our approach should also ease the correct formulation of program transformations.

3. Our approach also works for mutually recursive schemes, which are not currently available in PVS.

In [35], Farmer treats the definition of recursive functions in a logic of partial functions and represents schematic functions in a similar manner to our approach.

In the context of language design, Lewis *et al.* [63] use schemes to implement a degree of dynamic scoping in a statically scoped functional programming language. Their approach allows occurrences of a free variable, *e.g.*, $\mathcal{P}$, in the body of a program to be marked with special syntax, *e.g.*, $?\mathcal{P}$. The program is then treated as being parameterized by all such variables. To instantiate $\mathcal{P}$ occurring in a program f by a ground value val, they employ a notatation 'f `with` $?\mathcal{P} = val$'. Although their work is phrased using operational semantics and ours is denotationally based, there are many similarities.

Finally, our approach gives a higher-order and fully formal account of the *steadfast transformation* idea of Flener *et al.* [37]. In contrast to their work, we need give no soundness proof since our transformations are generated by deductive steps in a sound logic.

5.7 Call-by-name and call-by-value

Kapur and Subramaniam [56] pose the following induction challenge (we have slightly edited it for stylistic purposes). Consider a datatype of arithmetic expressions arith, having constructors for constants (C), variables (V), the addition of two expressions (Plus), and the Apply operator. The meaning of $\mathsf{Apply}\ B\ v\ M$ is $[v \mapsto M]B$, thus B is meant to be understood as a function body, v as a formal parameter, and M as the actual parameter.

$$\begin{array}{rcl}
\mathsf{C} & : & \mathsf{num} \to \alpha\ \mathsf{arith} \\
\mathsf{V} & : & \alpha \to \alpha\ \mathsf{arith} \\
\mathsf{Plus} & : & \alpha\ \mathsf{arith} \to \alpha\ \mathsf{arith} \to \alpha\ \mathsf{arith} \\
\mathsf{Apply} & : & \alpha\ \mathsf{arith} \to \alpha\ \mathsf{arith} \to \alpha\ \mathsf{arith} \to \alpha\ \mathsf{arith}
\end{array}$$

Two ways to evaluate expressions are given. The call-by-name strategy is a mutual (and nested) recursion with a 'helper' function:

$$\begin{array}{rcl}
\mathsf{CBN}\ (\mathsf{C}\ n, y, z) & \equiv & \mathsf{C}\ n \\
\mathsf{CBN}\ (\mathsf{V}\ x, y, z) & \equiv & \mathtt{if}\ x = y\ \mathtt{then}\ \mathsf{CBNh}\ z\ \mathtt{else}\ \mathsf{V}\ x \\
\mathsf{CBN}\ (\mathsf{Plus}\ a_1 a_2, y, z) & \equiv & \mathsf{Plus}\ (\mathsf{CBN}\ (a_1, y, z))(\mathsf{CBN}\ (a_2, y, z)) \\
\mathsf{CBN}\ (\mathsf{Apply}\ B\ v\ M, y, z) & \equiv & \mathsf{CBN}\ (\mathsf{CBN}\ (B, v, M), y, z)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{CBNh}\ (\mathsf{C}\ n) & \equiv & \mathsf{C}\ n \\
\mathsf{CBNh}\ (\mathsf{V}\ x) & \equiv & \mathsf{V}\ x \\
\mathsf{CBNh}\ (\mathsf{Plus}\ a_1 a_2) & \equiv & \mathsf{Plus}\ (\mathsf{CBNh}\ a_1)(\mathsf{CBNh}\ a_2) \\
\mathsf{CBNh}\ (\mathsf{Apply}\ B\ v\ M) & \equiv & \mathsf{CBN}\ (B, v, M)
\end{array}$$

The definition returns the given rules and the following induction theorem:

$$\begin{array}{l}
\forall P_0\ P_1. \\
\quad (\forall n\ y\ z. P_0\ (\mathsf{C}\ n, y, z)) \land \\
\quad (\forall x\ y\ z.((x = y) \supset P_1\ z) \supset P_0\ (\mathsf{V}\ x, y, z)) \land \\
\quad (\forall a_1\ a_2\ y\ z. P_0\ (a_2, y, z) \land P_0\ (a_1, y, z) \supset P_0\ (\mathsf{Plus}\ a_1\ a_2, y, z)) \land \\
\quad (\forall B\, v\, M\, y\, z. P_0(\mathsf{CBN}(B, v, M), y, z) \land P_0(B, v, M) \supset P_0(\mathsf{Apply}\, B\, v\, M, y, z)) \land \\
\quad (\forall n.\ P_1\ (\mathsf{C}\ n)) \land \\
\quad (\forall x.\ P_1\ (\mathsf{V}\ x)) \land \\
\quad (\forall a_1\ a_2\ . P_1\ a_2 \land P_1\ a_1 \supset P_1\ (\mathsf{Plus}\ a_1\ a_2)) \land \\
\quad (\forall B\ v\ M.\ P_0\ (B, v, M) \supset P_1\ (\mathsf{Apply}\ B\ v\ M)) \\
\qquad \supset \\
\quad (\forall v_0.\ P_0\ v_0) \land (\forall v_1. P_1\ v_1)
\end{array} \tag{5.6}$$

There are nine termination constraints, which we will abbreviate for clarity:

$$\begin{array}{l}
\mathsf{CBN\_Terminates}(R) \equiv \\
\quad \mathsf{WF}\ R\ \land \\
\quad (\forall M\ v\ B.\ R\ (\mathsf{INL}\ (B, v, M))\ (\mathsf{INR}\ (\mathsf{Apply}\ B\ v\ M)))\ \land \\
\quad (\forall a_1\ a_2.\ R\ (\mathsf{INR}\ a_2)\ (\mathsf{INR}\ (\mathsf{Plus}\ a_1\ a_2)))\ \land \\
\quad (\forall a_2\ a_1.\ R\ (\mathsf{INR}\ a_1)\ (\mathsf{INR}\ (\mathsf{Plus}\ a_1\ a_2)))\ \land \\
\quad (\forall z\ y\ M\ v\ B.\ R\ (\mathsf{INL}\ (\mathsf{auxCBN}\ R\ (\mathsf{INL}\ (B, v, M)), y, z)) \\
\qquad\qquad\qquad (\mathsf{INL}\ (\mathsf{Apply}\ B\ v\ M, y, z)))\ \land \\
\quad (\forall z\ y\ M\ v\ B.\ R\ (\mathsf{INL}\ (B, v, M))\ (\mathsf{INL}\ (\mathsf{Apply}\ B\ v\ M, y, z)))\ \land \\
\quad (\forall a_1\ z\ y\ a_2.\ R\ (\mathsf{INL}\ (a_2, y, z))\ (\mathsf{INL}\ (\mathsf{Plus}\ a_1\ a_2, y, z)))\ \land \\
\quad (\forall a_2\ z\ y\ a_1.\ R\ (\mathsf{INL}\ (a_1, y, z))\ (\mathsf{INL}\ (\mathsf{Plus}\ a_1\ a_2, y, z)))\ \land \\
\quad (\forall z\ y\ x.\ (x = y) \supset R\ (\mathsf{INR}\ z)\ (\mathsf{INL}\ (\mathsf{V}\ x, y, z))).
\end{array}$$

Note how the nested invocation of CBN has been transformed into auxCBN in the termination constraints.

The call-by-value strategy (also a nested function) uses an environment of evaluated expressions, accessed by a simple lookup function:

$$\begin{array}{rcl}
\mathsf{lookup}\ x\ [\,] & \equiv & \mathsf{0} \\
\mathsf{lookup}\ x\ ((y, z) :: rst) & \equiv & \texttt{if}\ x = y\ \texttt{then}\ z\ \texttt{else}\ \mathsf{lookup}\ x\ rst \\
\\
\mathsf{CBV}\ (\mathsf{C}\ n, env) & \equiv & n \\
\mathsf{CBV}\ (\mathsf{V}\ x, env) & \equiv & \mathsf{lookup}\ x\ env \\
\mathsf{CBV}\ (\mathsf{Plus}\ a_1 a_2, env) & \equiv & \mathsf{CBV}\ (a_1, env) + \mathsf{CBV}\ (a_2, env) \\
\mathsf{CBV}\ (\mathsf{Apply}\ B\ v\ M, env) & \equiv & \mathsf{CBV}\ (B, (v, \mathsf{CBV}\ (M, env)) :: env)
\end{array}$$

The following are the termination conditions of CBV; we again make a definition that encapsulates them:

$$\begin{array}{l}
\mathsf{CBV\_Terminates}(R) \equiv \\
\quad \mathsf{WF}\ R\ \land \\
\quad (\forall a_2\ env\ a_1.\ R\ (a_1, env)\ (\mathsf{Plus}\ a_1\ a_2, env))\ \land \\
\quad (\forall a_1\ env\ a_2.\ R\ (a_2, env)\ (\mathsf{Plus}\ a_1\ a_2, env))\ \land \\
\quad (\forall v\ B\ env\ M.\ R\ (M, env)\ (\mathsf{Apply}\ B\ v\ M, env))\ \land \\
\quad (\forall env\ M\ v\ B.\ R\ (B, (v, \mathsf{auxCBV}\ R\ (M, env)) :: env) \\
\qquad\qquad\qquad (\mathsf{Apply}\ B\ v\ M, env))
\end{array}$$

With the definitions finished, the goal can be stated:

$$\begin{array}{l}
\mathsf{CBN\_Terminates}(R) \wedge \mathsf{CBV\_Terminates}(R_1) \\
\quad\supset \\
\forall x\ y\ z\ env. \\
\quad \mathsf{CBV}\ (\mathsf{CBN}\ (x, y, z), env) \\
\qquad\qquad = \\
\quad \mathsf{CBV}\ (x, (y, \mathsf{CBV}\ (z, env)) :: env)
\end{array}$$

Proof. The proof will be by induction with (5.6), where the following instantiations are made:

$$\begin{array}{lll}
P_0 & \mapsto & \boldsymbol{\lambda}(x, y, z).\ \forall env.\ \ \mathsf{CBV}\ (\mathsf{CBN}\ (x, y, z), env) \\
& & \qquad\qquad\qquad\qquad = \\
& & \qquad\qquad\qquad \mathsf{CBV}(x, (y, \mathsf{CBV}\ (z, env)) :: env) \\
& & \\
P_1 & \mapsto & \boldsymbol{\lambda}z.\ \forall env.\ \mathsf{CBV}\ (\mathsf{CBNh}\ z, env) = \mathsf{CBV}\ (z, env).
\end{array}$$

The instantiation for P_0 just sets it to the goal at hand. The instantiation for P_1 however, has been suggested by Boulton's method [17] for finding induction predicates for mutual recursive functions. With these two instantiations, the proof is an anticlimax: it is simply induction followed by conditional rewriting with the definitions of CBN, CBNh, CBV, and lookup.
□

The interest of this example certainly isn't in the details of the proof, since that was surprisingly easy, given a correctly instantiated induction theorem. What is more of interest to us is how simple the whole exercise was. It took the author very little time to type the example in and get the proof. In a setting where termination had to be proved first, things would have taken much longer and the point of the exercise—to see how the induction proof worked—would have been like a mirage: visible but not attainable without a great deal of sweat.

5.8 Formal unification revisited

In this section we present the definition and verification of a nested unification algorithm first described, informally but in great detail, by Manna and Waldinger [65].[5] This has become a standard, although still challenging example. The verification we present is broken down into a progression of theories: association lists, terms, substitutions, and unifiers. The formalizations and proofs in these support theories are due to Larry Paulson and Martin Coen; they based their

[5] Manna and Waldinger actually *synthesized* the algorithm.

work directly on the paper of Manna and Waldinger. In order to have a self-contained presentation, we include the complete development; however, we omit the proofs of all theorems in the support theories. After that, all that is left is to verify the algorithm. Here is where we make our contribution, by giving a new and simpler proof of termination. This also leads to a simple proof of correctness.

Larry Paulson verified this unification algorithm in the middle of the 1980s, using Cambridge LCF [83]. His student Coen later redid much of the support theories in Isabelle/HOL [24] but was not able to define the algorithm itself, since Isabelle/HOL did not provide a strong enough recursion principle at that time. In [2] Sten Agerholm performed a domain-theory verification of this algorithm, working from Paulson's Cambridge LCF version. A Type Theory verification of the algorithm was performed by Rouyer [91], and some recent ones can be found in [18, 69]. The fact that this program is still a significant verification approximately 20 years after it first made its way into the literature (which was already 10 years after Robinson (re-)discovered unification) is worth pondering.

Remark. This example was carried out in Isabelle/HOL; thus, relations are represented by the type $(\alpha \times \alpha)\mathsf{set}$.

5.8.1 Association lists

Substitutions are implemented by lists of pairs, and thus a small theory of *association lists* (represented by the type $(\alpha * \beta)\mathsf{list}$) is formalized. The assoc function is easy to define with primitive recursion. An interesting aspect is how the natural partiality of this function is accommodated by making the invoker supply a default value in case lookup is not successful.

Definition 53 (assoc)

$$\begin{array}{rcl} \mathsf{assoc}\ v\ d\ [\,] & \equiv & d \\ \mathsf{assoc}\ v\ d\ ((a,b) :: al) & \equiv & \mathtt{if}\ v = a\ \mathtt{then}\ b\ \mathtt{else}\ \mathsf{assoc}\ v\ d\ al \end{array}$$

There is also an induction principle for association lists.

Theorem 54 (alist_induct)

$$P\,[\,] \land (\forall x\ y\ xs. P(xs) \supset P((x,y) :: xs)) \supset P(l).$$

5.8.2 Terms

We define a simplified set of terms. The major difference is that terms are binary trees instead of n-ary trees. This is not a significant limitation, as Paulson argues.

Definition 55 (Terms)

$$\begin{array}{rcl} \mathsf{Var} & : & \alpha \to \alpha\ \mathsf{uterm} \\ \mathsf{Const} & : & \alpha \to \alpha\ \mathsf{uterm} \\ \mathsf{Comb} & : & \alpha\ \mathsf{uterm} \to \alpha\ \mathsf{uterm} \to \alpha\ \mathsf{uterm} \end{array}$$

Definition 56 (Variables in a term. $\mathsf{vars\_of} : \alpha\ \mathsf{uterm} \to \alpha\ \mathsf{set}$**)**

$$\begin{array}{rcl}
\mathsf{vars\_of}\ (\mathsf{Var}\ v) & \equiv & \{v\} \\
\mathsf{vars\_of}\ (\mathsf{Const}\ c) & \equiv & \{\} \\
\mathsf{vars\_of}\ (\mathsf{Comb}\ M\ N) & \equiv & \mathsf{vars\_of}\ M \cup \mathsf{vars\_of}\ N
\end{array}$$

$vars\_var\_iff$	$(v \in \mathsf{vars\_of}(\mathsf{Var}(w))) = (w = v)$
$monotone\_vars\_of$	$\mathsf{vars\_of}\ M \cup \mathsf{vars\_of}\ N$ $\subseteq$ $\mathsf{vars\_of}(\mathsf{Comb}\ M\ P) \cup \mathsf{vars\_of}(\mathsf{Comb}\ N\ Q)$
$finite\_vars\_of$	$\mathsf{finite}(\mathsf{vars\_of}\ M)$

Definition 57 (Occurs check (infix). $<:\ : \alpha\ \mathsf{uterm} \to \alpha\ \mathsf{uterm} \to \mathsf{bool}$**)**

$$\begin{array}{rcl}
u <: (\mathsf{Var}\ v) & \equiv & \mathsf{False} \\
u <: (\mathsf{Const}\ c) & \equiv & \mathsf{False} \\
u <: (\mathsf{Comb}\ M\ N) & \equiv & (u = M \vee u = N \vee u <: M \vee u <: N)
\end{array}$$

Notice that this definition implies that the occurs check is actually a *proper* suboccurrence check: for example, it is not true that $\mathsf{Var}\ v <: \mathsf{Var}\ v$.

Definition 58 (Size of a term. $\mathsf{uterm\_size} : \alpha\ \mathsf{uterm} \to \mathsf{num}$**)**

$$\begin{array}{rcl}
\mathsf{uterm\_size}\ (\mathsf{Var}\ v) & \equiv & 0 \\
\mathsf{uterm\_size}\ (\mathsf{Const}\ c) & \equiv & 0 \\
\mathsf{uterm\_size}\ (\mathsf{Comb}\ M\ N) & \equiv & \mathsf{Suc}(\mathsf{uterm\_size}\ M + \mathsf{uterm\_size}\ N)
\end{array}$$

5.8.3 Substitutions

Substitutions are represented by association lists and applied by the following definition. We will employ the abbreviation $\alpha\ \mathsf{subst} \equiv (\alpha * \alpha\ \mathsf{uterm})\mathsf{list}$.

Definition 59 (Substitution (infix). $\lhd : \alpha\ \mathsf{uterm} \to \alpha\ \mathsf{subst} \to \alpha\ \mathsf{uterm}$**)**

$$\begin{array}{rcl}
(\mathsf{Var}\ v \lhd s) & \equiv & \mathsf{assoc}\ v\ (\mathsf{Var}\ v)\ s \\
(\mathsf{Const}\ c \lhd s) & \equiv & \mathsf{Const}\ c \\
(\mathsf{Comb}\ M\ N \lhd s) & \equiv & \mathsf{Comb}\ (M \lhd s)\ (N \lhd s)
\end{array}$$

subst_Nil	$t \triangleleft [\,] = t$
subst_mono	$(t <: u) \supset (t \triangleleft s <: u \triangleleft s)$
Var_not_occs	$\neg(\mathsf{Var}(v) <: t) \supset (t \triangleleft (v, t \triangleleft s)::s = t \triangleleft s)$
agreement	$(t \triangleleft r = t \triangleleft s) = \forall v.\ v \in \mathsf{vars\_of}(t) \supset \mathsf{Var}(v) \triangleleft r = \mathsf{Var}(v) \triangleleft s$
repl_invariance	$\neg v \in \mathsf{vars\_of}(t) \supset t \triangleleft (v, u)::s = t \triangleleft s$
Var_in_subst	$v \in \mathsf{vars\_of}(t) \supset w \in \mathsf{vars\_of}(t \triangleleft (v, \mathsf{Var}(w))::s)$
id_subst_lemma	$(M \triangleleft [(x,\ \mathsf{Var}\ x)]) = M$

Equality on substitutions

Substitutions are equal if no term can distinguish them.

Definition 60 (Equality (infix). $=_s$: α subst $\rightarrow$ α subst $\rightarrow$ bool)

$$(r =_s s) \equiv \forall t.\ t \triangleleft r = t \triangleleft s$$

subst_eq_iff	$(r =_s s) = \forall t.\ t \triangleleft r = t \triangleleft s$
subst_equiv	$r = s \supset r =_s s \ \wedge$ $r =_s s \supset s =_s r \ \wedge$ $q =_s r \supset r =_s s \supset q =_s s$
subst_subst2	$(r =_s s) \wedge P(t \triangleleft r)(u \triangleleft r) \supset P(t \triangleleft s)(u \triangleleft s)$
ssubst_subst2	$(s =_s r) \wedge P(t \triangleleft r)(u \triangleleft r) \supset P(t \triangleleft s)(u \triangleleft s)$
Cons_trivial	$(w, \mathsf{Var}(w) \triangleleft s)::s =_s s$

Composition of substitutions

Definition 61 ((infix) $\bullet$: α subst $\rightarrow$ α subst $\rightarrow$ α subst)

$$\begin{array}{rcl} [\,] \bullet bl & \equiv & bl \\ ((a, b)::al) \bullet bl & \equiv & (a, b \triangleleft bl)::(al \bullet bl) \end{array}$$

$comp\_Nil$	$s \bullet [\,] = s$
$subst\_comp$	$(t \lhd (r \bullet s)) = ((t \lhd r) \lhd s)$
$comp\_assoc$	$(q \bullet r) \bullet s =_s q \bullet (r \bullet s)$
$subst\_cong$	$(\theta_1 =_s \theta_2) \supset (\sigma_1 =_s \sigma_2) \supset ((\theta_1 \bullet \sigma_1) =_s (\theta_2 \bullet \sigma_2))$
$comp\_subst\_subst$	$(q \bullet r =_s s) \supset ((t \lhd q) \lhd r = t \lhd s)$

Domain and range of a substitution

Definition 62 (Domain. $\mathsf{sdom} : \alpha\ \mathsf{subst} \to \alpha\ \mathsf{set}$**)**

$$\begin{array}{rcl} \mathsf{sdom}\,[\,] & \equiv & \{\} \\ \mathsf{sdom}((a, b)::al) & \equiv & \texttt{if}\ \mathsf{Var}(a) = b\ \texttt{then}\ (\mathsf{sdom}\ al) - \{a\}\ \texttt{else}\ (\mathsf{sdom}\ al) \cup \{a\} \end{array}$$

Definition 63 (Range. $\mathsf{srange} : \alpha\ \mathsf{subst} \to \alpha\ \mathsf{set}$**)**

$$\mathsf{srange}(al) \equiv \bigcup \{y.\ \exists x \in \mathsf{sdom}(al).\ y = \mathsf{vars\_of}(\mathsf{Var}(x) \lhd al)\}$$

$sdom\_iff$	$(v \in \mathsf{sdom}(s)) = \neg(\mathsf{Var}(v) \lhd s = \mathsf{Var}(v))$
$srange\_iff$	$v \in \mathsf{srange}(s) = \exists w.\ w \in \mathsf{sdom}(s)$ $\wedge\ v \in \mathsf{vars\_of}(\mathsf{Var}(w) \lhd s)$
$invariance$	$(t \lhd s = t) = (\mathsf{sdom}(s) \cap \mathsf{vars\_of}(t) = \{\})$
Var_elim	$v \in \mathsf{sdom}(s) \supset \neg(v \in \mathsf{srange}(s)) \supset \neg v \in \mathsf{vars\_of}(t \lhd s)$
Var_elim2	$v \in \mathsf{sdom}(s) \wedge v \in \mathsf{vars\_of}(t \lhd s) \supset v \in \mathsf{srange}(s)$
Var_intro	$v \in \mathsf{vars\_of}(t \lhd s) \supset v \in \mathsf{srange}(s) \vee v \in \mathsf{vars\_of}(t)$
$srangeE$	$v \in \mathsf{srange}(s) \supset \exists w.\ w \in \mathsf{sdom}(s)$ $\wedge\ v \in \mathsf{vars\_of}(\mathsf{Var}(w) \lhd s)$
$dom\_range\_disjoint$	$(\mathsf{sdom}(s) \cap \mathsf{srange}(s) = \{\})$ $= \forall t.\ \mathsf{sdom}(s) \cap \mathsf{vars\_of}(t \lhd s) = \{\}$
$subst\_not\_empty$	$\neg(u \lhd s = u) \supset \exists x.\ x \in \mathsf{sdom}(s)$

More General Than

Definition 64 ((infix). $\gg : \alpha\ \mathsf{subst} \to \alpha\ \mathsf{subst} \to \mathsf{bool}$**)**

$$r \gg s \equiv \exists q.\ s =_s r \bullet q$$

$MoreGen\_iff$	$(r \gg s) = \exists q.\ s =_s (r \bullet q)$
$MoreGen\_Nil$	$[\,] \gg s$

5.8.4 Unifiers

Unifiers are substitutions that make terms equal. Most general unifiers can be instantiated to get any unifier.

Definition 65 (Unifier. : α subst $\rightarrow$ α uterm $\rightarrow$ α uterm $\rightarrow$ bool)

$$\mathsf{Unifier}\ s\ t\ u \equiv (t \triangleleft s = u \triangleleft s)$$

$Unifier\_Comb$	$\mathsf{Unifier}\ \theta\ (\mathsf{Comb}\ t\ u)\ (\mathsf{Comb}\ v\ w) \supset$ $\mathsf{Unifier}\ \theta\ t\ v \wedge \mathsf{Unifier}\ \theta\ u\ w$
$Cons\_Unifier$	$\neg v \in \mathsf{vars\_of}(t) \wedge \neg v \in \mathsf{vars\_of}(u) \wedge \mathsf{Unifier}\ s\ t\ u$ $\supset \mathsf{Unifier}\ ((v, r){::}s)\ t\ u$

Definition 66 (MGU. α subst $\rightarrow$ α uterm $\rightarrow$ α uterm $\rightarrow$ bool)

$$\mathsf{MGU}\ s\ t\ u \equiv \mathsf{Unifier}\ s\ t\ u \wedge \forall r.\ \mathsf{Unifier}\ r\ t\ u \supset s \gg r$$

MGU_iff	$\mathsf{MGU}\ s\ t\ u = \forall r.\ \mathsf{Unifier}\ r\ t\ u = s \gg r$
$MGUnifier\_Var$	$\neg\mathsf{Var}(v) <: t \supset \mathsf{MGU}[(v, t)]\ (\mathsf{Var}\ v)\ t$
mgu_sym	$\mathsf{MGU}\ s\ t\ u = \mathsf{MGU}\ s\ u\ t$

5.8.5 Unify

Now we give the unification algorithm.

Definition 67 (**Unify**: (α uterm $*$ α uterm) $\rightarrow$ α subst result)

```
Unify(Const $m$, Const $n$) = if ($m = n$) then Some[] else None
Unify(Const $m$, Comb $M$ $N$) = None
Unify(Const $m$, Var $v$) = Some[($v$, Const $m$)]
Unify(Var $v$, $M$) = if (Var $v$ <: $M$) then None else Some[($v$, $M$)]
Unify(Comb $M$ $N$, Const $x$) = None
Unify(Comb $M$ $N$, Var $v$) =  if (Var $v$ <: Comb $M$ $N$) then None
                                 else Some[($v$, Comb $M$ $N$)]
Unify(Comb $M_1$ $N_1$, Comb $M_2$ $N_2$) =
      case Unify($M_1$, $M_2$)
        of None => None
         | Some $\theta$ => case Unify($N_1 \lhd \theta, N_2 \lhd \theta$)
                             of None => None
                              | Some $\sigma$ => Some($\theta \bullet \sigma$)
```

Termination relation

The termination relation for the unification algorithm operates on pairs of terms and checks first for a proper subset relation on the variables and then for smaller sizes. These are put together by mapping into a lexicographic combination with an inverse image. The second element in the compound relation is exactly the relational product of the size measure, but that is subsumed in the lexicographic combination.

Definition 68 (UTR : ((α uterm $*$ α uterm) $*$ (α uterm $*$ α uterm))set)

$$\begin{aligned}\mathsf{UTR} \equiv \mathsf{inv\_image}\quad & (\mathsf{LEX\ fpss\ (LEX\ (measure\ uterm\_size)\ (measure\ uterm\_size))})\\ & ((\boldsymbol{\lambda}(x, y).\ \mathsf{vars\_of}\ x \cup \mathsf{vars\_of}\ y)\#(\boldsymbol{\lambda} x.\ x))\end{aligned}$$

When the recursion equations together with UTR are submitted to TFL, the following results are returned: a constrained set of equations, a constrained induction principle, and the termination conditions (the constraints).

Theorem 69 (Unify - output)

1. $\mathsf{Unify}(\mathsf{Const}\ m, \mathsf{Const}\ n) = \mathtt{if}\ m = n\ \mathtt{then}\ \mathsf{Some}[\,]\ \mathtt{else}\ \mathsf{None}$
2. $\mathsf{Unify}(\mathsf{Const}\ m, \mathsf{Comb}\ M\ N) = \mathsf{None}$
3. $\mathsf{Unify}(\mathsf{Const}\ m, \mathsf{Var}\ v) = \mathsf{Some}[(v, \mathsf{Const}\ m)]$
4. $\mathsf{Unify}(\mathsf{Var}\ v, M) = \mathtt{if}\ (\mathsf{Var}\ v <: M)\ \mathtt{then}\ \mathsf{None}\ \mathtt{else}\ \mathsf{Some}[(v, M)]$
5. $\mathsf{Unify}(\mathsf{Comb}\ M\ N, \mathsf{Const}\ x) = \mathsf{None}$
6. $\mathsf{Unify}(\mathsf{Comb}\ M\ N, \mathsf{Var}\ v) = \begin{array}[t]{l} \mathtt{if}\ (\mathsf{Var}\ v <: \mathsf{Comb}\ M\ N)\ \mathtt{then}\ \mathsf{None} \\ \mathtt{else}\ \mathsf{Some}[(v, \mathsf{Comb}\ M\ N)] \end{array}$
7. $$\left[\begin{array}{l} \mathsf{WF\ UTR}, \\ ((M_1, M_2), (\mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2)) \in \mathsf{UTR}, \\ \forall \theta.\ \mathsf{Unify}(M_1, M_2) = \mathsf{Some}\ \theta \\ \qquad \supset ((N_1 \lhd \theta, N_2 \lhd \theta), (\mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2)) \in \mathsf{UTR} \end{array}\right]$$
$$\begin{array}{l} \vdash \\ \mathsf{Unify}(\mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2) = \\ \qquad \mathtt{case}\ \mathsf{Unify}(M_1, M_2) \\ \qquad\quad \mathtt{of}\ \mathsf{None} => \mathsf{None} \\ \qquad\quad\ \mid \mathsf{Some}\ \theta => \\ \qquad\qquad\qquad \mathtt{case}\ \mathsf{Unify}(N_1 \lhd \theta, N_2 \lhd \theta) \\ \qquad\qquad\qquad\quad \mathtt{of}\ \mathsf{None} => \mathsf{None} \\ \qquad\qquad\qquad\quad\ \mid \mathsf{Some}\ \sigma => \mathsf{Some}(\theta \bullet \sigma) \end{array}$$

Our goal is to prove and eliminate the termination constraints on clause 7. The wellfoundedness of UTR, and also the inner termination condition, can be proved straightforwardly and eliminated. However, the nested recursive call is not so simple. To help in proving it, we will use the following induction theorem which our machinery has automatically proved.

Theorem 70 (Induction - output)

$$\left[\begin{array}{l} \mathsf{WF\ UTR}, \\ ((M_1, M_2), \mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2) \in \mathsf{UTR} \end{array}\right]$$
$$\begin{array}{l} \ \ \vdash \\ \forall P.\ (\forall m\ n.\ P\ (\mathsf{Const}\ m, \mathsf{Const}\ n))\ \wedge \\ \qquad (\forall m\ M\ N.\ P\ (\mathsf{Const}\ m, \mathsf{Comb}\ M\ N))\ \wedge \\ \qquad (\forall m\ v.\ P\ (\mathsf{Const}\ m, \mathsf{Var}\ v))\ \wedge \\ \qquad (\forall v\ M.\ P\ (\mathsf{Var}\ v, M))\ \wedge \\ \qquad (\forall M\ N\ x.\ P\ (\mathsf{Comb}\ M\ N, \mathsf{Const}\ x))\ \wedge \\ \qquad (\forall M\ N\ v.\ P\ (\mathsf{Comb}\ M\ N, \mathsf{Var}\ v))\ \wedge \\ \qquad (\forall M_1\ N_1\ M_2\ N_2. \\ \qquad\quad (\forall \theta.\ \mathsf{Unify}(M_1, M_2) = \mathsf{Some}\ \theta \\ \qquad\qquad\quad \supset ((N_1 \lhd \theta, N_2 \lhd \theta), (\mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2)) \in \mathsf{UTR} \\ \qquad\qquad\qquad \supset P\ (N_1 \lhd \theta, N_2 \lhd \theta)) \\ \qquad\quad \wedge\ P\ (M_1, M_2) \supset P\ (\mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2)) \\ \qquad\quad \supset \\ \qquad \forall v\ v_1.\ P\ (v, v_1) \end{array}$$

5.8.6 Termination

Theorem 71 (UTR is wellfounded) $\mathsf{WF\ UTR}$

Proof. fpss is wellfounded. The variables in a term are finite. The union of two finite sets is finite. All measure functions are wellfounded. Lexicographic product and inverse image both propagate wellfoundedness. □.

Now we prove the termination relation applied to the inner call:

Theorem 72 (Inner termination condition)

$$((M_1, M_2),\ (\mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2)) \in \mathsf{UTR}$$

Proof. Either N_1 or N_2 introduce new variables or they don't. If they do, we apply fpss, the first projection of the termination relation. If they don't then we apply the second projection: M_1 is smaller than Comb M_1 N_1 and M_2 is smaller than Comb M_2 N_2. □

We eliminate these two constraints from the rules and the induction theorem. In the proof of the nested termination condition, we will need the following lemma in several of the base cases. It is a clumsy formulation (requiring two conjuncts, each with exactly the same proof) of a more general theorem, which we do not prove.

Theorem 73 (var_elimR)

$$\begin{array}{l}(\neg \mathsf{Var}\ x <: M) \wedge ([(x, M)] = \theta) \supset \\ \quad \forall N_1\ N_2.\ (((N_1 \lhd \theta, N_2 \lhd \theta), (\mathsf{Comb}\ M\ N_1, \mathsf{Comb}\ (\mathsf{Var}\ x)\ N_2)) \in \mathsf{UTR}) \\ \qquad\qquad\quad \wedge\ (((N_1 \lhd \theta, N_2 \lhd \theta), (\mathsf{Comb}(\mathsf{Var}\ x)\ N_1, \mathsf{Comb}\ M\ N_2)) \in \mathsf{UTR})\end{array}$$

At the very last step of the termination proof for the nested call, we need the following lemma about UTR. Loosely, it says that UTR ignores a certain amount of term structure. We also need the fact that UTR is transitive.

Theorem 74 (UTRassoc)

$$\begin{array}{c}((X, Y), (\mathsf{Comb}\ A\ (\mathsf{Comb}\ B\ C), \mathsf{Comb}\ D\ (\mathsf{Comb}\ E\ F))) \in \mathsf{UTR} \\ \supset \\ ((X, Y), (\mathsf{Comb}\ (\mathsf{Comb}\ A\ B)\ C, \mathsf{Comb}\ (\mathsf{Comb}\ D\ E)\ F)) \in \mathsf{UTR}.\end{array}$$

Theorem 75 (UTRtrans) transitive UTR

Now, finally, we are at the crux of our development.

Theorem 76 (Nested termination condition)

$$\begin{array}{l}\mathsf{Unify}(M_1, M_2) = \mathsf{Some}\ \theta \\ \quad \supset \\ ((N_1 \lhd \theta, N_2 \lhd \theta), (\mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2)) \in \mathsf{UTR}\end{array}$$

Proof. The first thing we do is restrict the scopes of N_1 and N_2. Thus it suffices to prove

$$\begin{array}{l}\forall M_1\ M_2\ \theta.\ \mathsf{Unify}(M_1, M_2) = \mathsf{Some}\ \theta \supset \\ \qquad\qquad \forall N_1\ N_2.\ ((N_1 \lhd \theta, N_2 \lhd \theta), (\mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2)) \in \mathsf{UTR}\end{array}$$

Now we induct with Theorem 70 and simplify with the rewrite rules for Unify. (It turns out that in this proof, we will only need the *non-recursive* rewrite rules, *i.e.*clauses 1-6 of Theorem 69.) This eliminates the cases where Unify can only fail: 2 and 5. Of the remaining base cases, Case 1 is easy to check, and Cases 3, 4, and 6 are all instances of theorem *var_elimR*.

1. $((N_1 \lhd [\,], N_2 \lhd [\,]), \mathsf{Comb}(\mathsf{Const}\ n)\ N_1, \mathsf{Comb}\ (\mathsf{Const}\ n)\ N_2) \in \mathsf{UTR}.$

3. $\begin{array}{l}((N_1 \lhd [(x, \mathsf{Const}\ m)], N_2 \lhd [(x, \mathsf{Const}\ m)]), \\ (\mathsf{Comb}\ (\mathsf{Const}\ m)\ N_1, \mathsf{Comb}\ (\mathsf{Var}\ x)\ N_2)) \in \mathsf{UTR}\end{array}$

4. $\begin{array}{l}\neg(\mathsf{Var}\ x <: M) \wedge ([(x, M)] = \mathit{theta}) \ \supset \\ ((N_1 \lhd \theta, N_2 \lhd \theta), (\mathsf{Comb}\ (\mathsf{Var}\ x)\ N_1, \mathsf{Comb}\ M\ N_2)) \in \mathsf{UTR}\end{array}$

6. $\begin{array}{l}\neg(\mathsf{Var}\ x <: \mathsf{Comb}\ M\ N) \ \supset \\ \quad ((N_1 \lhd [(x, \mathsf{Comb}\ M\ N)], N_2 \lhd [(x, \mathsf{Comb}\ M\ N)]), \\ \quad (\mathsf{Comb}\ (\mathsf{Comb}\ M\ N)\ N_1, \mathsf{Comb}\ (\mathsf{Var}\ x)\ N_2)) \in \mathsf{UTR}\end{array}$

This leaves the recursive case:

$$\begin{array}{l}\left(\begin{array}{l}\mathtt{case}\ \mathsf{Unify}(M_1, M_2) \\ \quad \mathtt{of}\ \mathsf{None} => \mathsf{None} \\ \quad\ \mid \mathsf{Some}\ \theta_3 => \mathtt{case}\ (\mathsf{Unify}(N_1 \lhd \theta_3, N_2 \lhd \theta_3) \\ \qquad\qquad\qquad\qquad \mathtt{of}\ \mathsf{None} => \mathsf{None} \\ \qquad\qquad\qquad\qquad\ \mid \mathsf{Some}\ \sigma => \mathsf{Some}(\theta_3 \bullet \sigma))\end{array}\right) = \mathsf{Some}\ \theta \\ \quad \supset \\ \forall P\ Q.((P \lhd \theta, Q \lhd \theta), \mathsf{Comb}(\mathsf{Comb}\ M_1\ N_1)\ P, \mathsf{Comb}(\mathsf{Comb}\ M_2\ N_2)\ Q) \in \mathsf{UTR}\end{array}$$

The two inductive hypotheses are

$$\begin{array}{l}\forall \theta_0.\ \ \mathsf{Unify}(M_1, M_2) = \mathsf{Some}\ \theta_0 \supset \\ \qquad ((N_1 \lhd \theta_0, N_2 \lhd \theta_0), (\mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2)) \in \mathsf{UTR} \supset \\ \qquad\quad \forall \theta_1.\ \mathsf{Unify}(N_1 \lhd \theta_0, N_2 \lhd \theta_0) = \mathsf{Some}\ \theta_1 \supset \\ \qquad\qquad \forall P\ Q.\ ((P \lhd \theta_1, Q \lhd \theta_1), (\mathsf{Comb}(N_1 \lhd \theta_0) P, \mathsf{Comb}(N_2 \lhd \theta_0) Q)) \in \mathsf{UTR}\end{array} \tag{5.7}$$

and

$$\begin{array}{l}\forall \theta_2.\ \ \mathsf{Unify}(M_1, M_2) = \mathsf{Some}\ \theta_2\ \supset \\ \qquad \forall N_1\ N_2.\ ((N_1 \lhd \theta_2, N_2 \lhd \theta_2), (\mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2)) \in \mathsf{UTR}.\end{array} \tag{5.8}$$

Making a case analysis on $\mathsf{Unify}(M_1, M_2)$, we see that the None case is vacuously true; alternatively, suppose that $\mathsf{Some}\ \theta_3$ is the result. We can apply the second induction hypothesis to prove

$$\forall N_1\ N_2.\ ((N_1 \lhd \theta_3, N_2 \lhd \theta_3), (\mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2)) \in \mathsf{UTR} \tag{5.9}$$

and therefore the first induction hypothesis can be simplified (twice!) to obtain

$$\begin{array}{l} \forall \theta_1.\ \mathsf{Unify}(N_1 \lhd \theta_3, N_2 \lhd \theta_3) = \mathsf{Some}\ \theta_1 \supset \\ \qquad \forall P\ Q.\ ((P \lhd \theta_1, Q \lhd \theta_1), (\mathsf{Comb}(N_1 \lhd \theta_3)P, \mathsf{Comb}(N_2 \lhd \theta_3)Q)) \in \mathsf{UTR}. \end{array} \tag{5.10}$$

and the goal twice to get the following goal:

$$\begin{array}{l} \left(\begin{array}{l} \mathtt{case}\ (\mathsf{Unify}(N_1 \lhd \theta_3, N_2 \lhd \theta_3)) \\ \quad \mathtt{of}\ \mathsf{None} \Rightarrow \mathsf{None} \\ \quad \mid \mathsf{Some}\ \sigma \Rightarrow \mathsf{Some}(\theta_3 \bullet \sigma) \end{array} \right) = \mathsf{Some}\ \theta \\ \supset \\ \left(\begin{array}{l} (P \lhd \theta, Q \lhd \theta), \\ (\mathsf{Comb}(\mathsf{Comb}\ M_1\ N_1)\ P, \mathsf{Comb}(\mathsf{Comb}\ M_2\ N_2)\ Q) \end{array} \right) \in \mathsf{UTR}. \end{array}$$

We now make a case analysis on $\mathsf{Unify}(N_1 \lhd \theta_3, N_2 \lhd \theta_3)$. Again, the None case is vacuously true, and so suppose that $\mathsf{Some}\ \sigma$ is the result. We are left with the goal

$$\begin{array}{l} (\mathsf{Some}(\theta_3 \bullet \sigma) = \mathsf{Some}\ \theta) \\ \supset \\ \left(\begin{array}{l} (P \lhd \theta, Q \lhd \theta), \\ (\mathsf{Comb}(\mathsf{Comb}\ M_1\ N_1)\ P, \mathsf{Comb}(\mathsf{Comb}\ M_2\ N_2)\ Q) \end{array} \right) \in \mathsf{UTR}, \end{array}$$

and hence the goal (by use of the injectivity of Some and also the theorem *subst_comp*)

$$\left(\begin{array}{l} (P \lhd \theta_3 \lhd \sigma, Q \lhd \theta_3 \lhd \sigma), \\ (\mathsf{Comb}(\mathsf{Comb}\ M_1\ N_1)\ P, \mathsf{Comb}(\mathsf{Comb}\ M_2\ N_2)\ Q) \end{array} \right) \in \mathsf{UTR}.$$

We can also further simplify (5.10) to get

$$\forall P\ Q.\ \left(\begin{array}{l} (P \lhd \sigma, Q \lhd \sigma), \\ (\mathsf{Comb}(N_1 \lhd \theta_3)P, \mathsf{Comb}(N_2 \lhd \theta_3)Q) \end{array} \right) \in \mathsf{UTR}. \tag{5.11}$$

We can then instantiate (5.11) by $P \mapsto P \lhd \theta_3$ and $Q \mapsto Q \lhd \theta_3$ to prove

$$\left(\begin{array}{l} (P \lhd \theta_3 \lhd \sigma, Q \lhd \theta_3 \lhd \sigma), \\ (\mathsf{Comb}(N_1 \lhd \theta_3)(P \lhd \theta_3), \mathsf{Comb}(N_2 \lhd \theta_3)(Q \lhd \theta_3)) \end{array} \right) \in \mathsf{UTR}. \tag{5.12}$$

By the definition of substitution, this is equal to

$$\left(\begin{array}{l} (P \lhd \theta_3 \lhd \sigma, Q \lhd \theta_3 \lhd \sigma), \\ (\mathsf{Comb}\ N_1\ P \lhd \theta_3, \mathsf{Comb}\ N_2\ Q \lhd \theta_3) \end{array} \right) \in \mathsf{UTR}. \tag{5.13}$$

This allows us to use the fact that UTR is transitive to reduce the goal to showing

$$\begin{pmatrix} (\mathsf{Comb}\ N_1\ P \lhd \theta_3, \mathsf{Comb}\ N_2\ Q \lhd \theta_3), \\ (\mathsf{Comb}(\mathsf{Comb}\ M_1\ N_1)\ P, \mathsf{Comb}(\mathsf{Comb}\ M_2\ N_2)\ Q) \end{pmatrix} \in \mathsf{UTR}.$$

We can now instantiate (5.9) with $N_1 \mapsto \mathsf{Comb}\ N_1\ P$ and $N_2 \mapsto \mathsf{Comb}\ N_2\ Q$ to prove

$$\begin{pmatrix} (\mathsf{Comb}\ N_1\ P \lhd \theta_3, \mathsf{Comb}\ N_2\ Q \lhd \theta_3), \\ (\mathsf{Comb}\ M_1\ (\mathsf{Comb}\ N_1\ P), \mathsf{Comb}\ M_2\ (\mathsf{Comb}\ N_2\ Q)) \end{pmatrix} \in \mathsf{UTR}.$$

UTRassoc now applies.
□

With termination proved, we finally obtain a pristine set of rules and induction theorem:

Theorem 77 (Unify - final rules)

```
$\mathsf{Unify}(\mathsf{Const}\ m, \mathsf{Const}\ n) = \mathit{if}\ (m = n)\ \mathit{then}\ \mathsf{Some}[\,]\ \mathit{else}\ \mathsf{None}$
$\mathsf{Unify}(\mathsf{Const}\ m, \mathsf{Comb}\ M\ N) = \mathsf{None}$
$\mathsf{Unify}(\mathsf{Const}\ m, \mathsf{Var}\ v) = \mathsf{Some}[(v, \mathsf{Const}\ m)]$
$\mathsf{Unify}(\mathsf{Var}\ v, M) =$ if $(\mathsf{Var}\ v <: M)$ then $\mathsf{None}$ else $\mathsf{Some}[(v, M)]$
$\mathsf{Unify}(\mathsf{Comb}\ M\ N, \mathsf{Const}\ x) = \mathsf{None}$
$\mathsf{Unify}(\mathsf{Comb}\ M\ N, \mathsf{Var}\ v) =$  if $(\mathsf{Var}\ v <: \mathsf{Comb}\ M\ N)$ then $\mathsf{None}$
                                  else $\mathsf{Some}[(v, \mathsf{Comb}\ M\ N)]$
$\mathsf{Unify}(\mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2) =$
    case $\mathsf{Unify}(M_1, M_2)$
     of $\mathsf{None} => \mathsf{None}$
      | $\mathsf{Some}\ \theta =>$ case $\mathsf{Unify}(N_1 \lhd \theta, N_2 \lhd \theta)$
                       of $\mathsf{None} => \mathsf{None}$
                        | $\mathsf{Some}\ \sigma => \mathsf{Some}(\theta \bullet \sigma)$
```

Theorem 78 (Induction - final)

$$\begin{aligned}
\forall P.\quad & (\forall m\, n.\ P\ (\mathsf{Const}\ m, \mathsf{Const}\ n)) \wedge \\
& (\forall m\ M\ N.\ P\ (\mathsf{Const}\ m, \mathsf{Comb}\ M\ N)) \wedge \\
& (\forall m\ v.\ P\ (\mathsf{Const}\ m, \mathsf{Var}\ v)) \wedge \\
& (\forall v\ M.\ P\ (\mathsf{Var}\ v, M)) \wedge \\
& (\forall M\ N\ x.\ P\ (\mathsf{Comb}\ M\ N, \mathsf{Const}\ x)) \wedge \\
& (\forall M\ N\ v.\ P\ (\mathsf{Comb}\ M\ N, \mathsf{Var}\ v)) \wedge \\
& (\forall M_1\ N_1\ M_2\ N_2. \\
& \quad (\forall \theta.\ \mathsf{Unify}(M_1, M_2) = \mathsf{Some}\ \theta \supset P\ (N_1 \lhd \theta, N_2 \lhd \theta)) \wedge P\ (M_1, M_2) \\
& \qquad \supset P\ (\mathsf{Comb}\ M_1\ N_1, \mathsf{Comb}\ M_2\ N_2)) \\
& \supset \forall v\ v_1.\ P\ (v, v_1)
\end{aligned}$$

5.8.7 Correctness

Armed with these, the correctness of Unify can be established. Unlike Manna and Waldinger, we don't need idempotence to prove that the algorithm produces most-general unifiers. (They used idempotence at a crucial point in their termination proof.) This dramatically simplifies the proof of correctness.

Theorem 79 (Unify_gives_MGU)

$$\forall\theta.\ \mathsf{Unify}(P,Q) = \mathsf{Some}\ \theta \supset \mathsf{MGU}\ \theta\ P\ Q$$

Proof. By induction using Theorem 78. The base cases are either trivial, or use the lemma *MGUnifier_Var*. In the recursive case, the failure branches are trivial. Thus assume that $\mathsf{Unify}(M_1, M_2) = \mathsf{Some}\ \theta$ and $\mathsf{Unify}(N_1 \lhd \theta, N_2 \lhd \theta) = \mathsf{Some}\ \sigma$ hold. Also assume

$$\begin{array}{ll} \mathsf{MGU}\ \sigma\ (N_1 \lhd \theta)\ (N_2 \lhd \theta) & \text{and} \\ \mathsf{MGU}\ \theta\ M_1\ M_2 & \end{array}$$

It remains to show $\mathsf{MGU}(\theta \bullet \sigma)\ (\mathsf{Comb}\ M_1\ N_1)\ (\mathsf{Comb}\ M_2\ N_2)$. It is immediate from the definitions of MGU and Unifier and the theorem *subst_comp* that $\theta \bullet \sigma$ unifies $(\mathsf{Comb}\ M_1\ N_1)$ and $(\mathsf{Comb}\ M_2\ N_2)$; we now show that $\theta\bullet\sigma$ is most general. Assume that γ unifies $(\mathsf{Comb}\ M_1\ N_1)$ and $(\mathsf{Comb}\ M_2\ N_2)$, *i.e.*,

$$\begin{array}{ll} M_1 \lhd \gamma = M_2 \lhd \gamma & \text{and} \\ N_1 \lhd \gamma = N_2 \lhd \gamma. & \end{array}$$

There is a δ such that $\gamma =_s \theta \bullet \delta$ because θ is most general. By virtue of this, $(N_1 \lhd \theta) \lhd \delta = (N_2 \lhd \theta) \lhd \delta$. Thus there is a ρ such that $\delta =_s \sigma \bullet \rho$ because σ is most general. We wish to prove $\exists q.\ \gamma =_s \theta \bullet \sigma \bullet q$, and do so as follows:

$$\begin{array}{ll} & \exists q.\ \gamma =_s \theta \bullet \sigma \bullet q \\ = & \exists q.\ \theta \bullet \delta =_s \theta \bullet \sigma \bullet q \\ = & \exists q.\ \theta \bullet \delta =_s \theta \bullet (\sigma \bullet q) \\ = & \exists q.\ \delta =_s \sigma \bullet q \\ = & \exists q.\ \sigma \bullet \rho =_s \sigma \bullet q \\ = & \exists q.\ \rho =_s q \end{array}$$

□

Idempotence

The correctness of some applications of unification depend on the fact that the most general unifier that is returned is also *idempotent.* This fact is quite simple to establish, once a small support theory of idempotent substitutions is at hand.

Definition 80 ($\mathsf{Idem} : \alpha\ \mathsf{subst} \rightarrow \mathsf{bool}$)

$$\mathsf{Idem}(s) \equiv (s \bullet s) =_s s$$

Idem_Nil	$\mathsf{Idem}[\,]$
Idem_iff	$\mathsf{Idem}(s) = (\mathsf{sdom}(s) \cap \mathsf{srange}(s) = \{\})$
Var_Idem	$\neg(\mathsf{Var}(v) <: t) \supset \mathsf{Idem}[(v, t)]$
Unifier_Idem_subst	$\mathsf{Idem}(r) \wedge \mathsf{Unifier}\ s\ (t \lhd r)\ (u \lhd r)$ $\quad\supset \mathsf{Unifier}\ (r \bullet s)\ (t \lhd r)\ (u \lhd r)$
Idem_comp	$\mathsf{Idem}(r)$ $\quad\supset \mathsf{Unifier}\ s\ (t \lhd r)\ (u \lhd r)$ $\quad\quad\supset (\forall q.\ \mathsf{Unifier}\ q\ (t \lhd r)\ (u \lhd r) \supset (s \bullet q) =_s q)$ $\quad\quad\quad\supset \mathsf{Idem}(r \bullet s)$

Theorem 81 (Unify_gives_Idem)

$$\forall\theta.\mathsf{Unify}(P, Q) = \mathsf{Some}\ \theta \supset \mathsf{Idem}\ \theta$$

Proof. By induction with Theorem 78. As before, the base cases are either trivial, or yield to application of a single lemma (in this case *Var_Idem*). In the recursive case, assume $\mathsf{Unify}(M_1, M_2) = \mathsf{Some}\ \theta$ and $\mathsf{Unify}(N_1 \lhd \theta, N_2 \lhd \theta) = \mathsf{Some}\ \sigma$. Thus, by Theorem 79,

$$\begin{array}{l}\mathsf{MGU}\ \sigma\ (N_1 \lhd \theta)\ (N_2 \lhd \theta)\\ \mathsf{MGU}\ \theta\ M_1\ M_2.\end{array}$$

Assume $\mathsf{Idem}\ \sigma$ and $\mathsf{Idem}\ \theta$; it remains to show $\mathsf{Idem}\ (\theta \bullet \sigma)$. By *Idem_comp*, it suffices to show

$$\forall\gamma.\ \mathsf{Unifier}\ \gamma\ (N_1 \lhd \theta)\ (N_2 \lhd \theta) \supset \sigma \bullet \gamma =_s \gamma.$$

Assume γ unifies $(N_1 \lhd \theta)$ and $(N_2 \lhd \theta)$. There is a δ such that $\gamma =_s \sigma \bullet \delta$ because σ is an MGU of $(N_1 \lhd \theta)$ and $(N_2 \lhd \theta)$. Now

$$\begin{array}{rl} & \sigma \bullet \gamma =_s \gamma\\ = & \sigma \bullet (\sigma \bullet \delta) =_s \sigma \bullet \delta\\ = & (\sigma \bullet \sigma) \bullet \delta =_s \sigma \bullet \delta\\ = & \sigma \bullet \delta =_s \sigma \bullet \delta\end{array}$$

□

Chapter 6

Conclusions and Future Work

TFL is a sound extension to whatever HOL implementation it is instantiated to. Thus it can be thought of as a portable and safe definition mechanism for higher order recursive functions or, more grandiosely, as a re-usable deduction-based environment for *strong* functional programming [101]. To our knowledge, there are no other such proof tools in existence. However, we think that there should be, for progress in our field is impeded by the current replication of work. The same facilities get built over and over: datatype definition packages, automated reasoners, simplifiers, *etc.* Building mechanized proof tools is a demanding pursuit and the sharing of the tool-building work may allow better overall progress to be made.

Transferring our attention from the state of the research field to the system we have produced, the power and utility of TFL is based upon its novel combination of well-known theorems (wellfounded induction and recursion, consequences of logical datatypes) and algorithms (pattern matching and contextual rewriting) in higher order logic.

Our research progressed in stages: once the basic framework (function definitions made with a wellfounded relation) was realized, we wanted to make life simpler for a user; after some time, we eventually saw how non-nested relationless definitions could be achieved. Once they were in place, there was the irritating restriction of non-nestedness to overcome in relationless definitions. For a long while, we were blocked on this problem, but then we saw how relationless definitions supported schematic definitions (a separate thread of enquiry). That lead to the realization that schemes could be employed to implement relationless nested definitions. After that, the extension to mutual recursive definitions (relationless or not) was quick once we realized, after reading Boulton's paper, what the shape of the required induction theorems should be. Thus the initial platform built from existing technology allowed new problems to be posed and solved, resulting in a much more general system than initially envisaged.

The major new outcomes of this work are the following:

- Direct definitions where the termination relation is supplied at definition time. Mutual and nested recursions are supported. (By a *direct* definition, we mean that the specified recursion equations will automatically be derived as HOL theorems, along with an induction theorem (and of course, any remaining termination constraints).)

- Direct definitions where the termination relation is not supplied. Mutual and nested recursions are supported. Thus proving termination need no longer provide an initial barrier to a correctness proof. As a consequence, this provides a secure and flexible basis for large-scale automation of termination proofs.

- Sound (and direct) program schemes (mutual and nested recursions are supported). We have shown how recursive definitions with parameters, *i.e.*, free variables can be automated. Although schemes may also be defined 'by hand', experience shows that the definition is contorted, and significant work must be done to achieve the desired recursion equations and induction theorem (Tobias Nipkow, personal communication). From the point of view of the main stream of work in program transformation, the interest of our schemes is their soundness: the termination constraints that automatically arise in our uniform treatment enforce the totality of any satisfiable instantiation of the scheme.

  From the standpoint of reasoning about schemes, a policy of jointly instantiating a scheme and its induction theorem ensures that the induction theorem is usable for reasoning about properties of the scheme, at any step in the instantiation chain, from initial scheme to final concrete program.

- We have deepened our understanding of termination proofs for nested recursive functions. In particular, a nested function can be used to prove its own termination, provided that it is used at 'smaller instances'. The heretofore standard practice of proving termination and partial correctness simultaneously is thus revealed to be a form of 'strengthening the induction hypothesis'. By example, we have shown several well-known nested algorithms where this strengthening is unnecessary, in contrast to statements in the literature. The benefit of our improved understanding is that termination proofs of nested recursive functions have become less mysterious. Giesl has made the same observation, although his solution is more meta-theoretic in flavour, and is in a more restricted logic.

Future work.

Our work raises several topics for investigation, which range from small projects that would only take a few months, to major research efforts.

Comparison with other definition approaches

Another way to define a recursive function would be to define an inductive relation representing the input/output pairs of the intended function. Then a proof of the *functionality* of the relation would be needed before the desired function could be defined. The advantage of the inductive definition approach is that it implicitly encodes the finiteness of the recursive call chains, so explicit consideration of wellfounded relations may be avoidable. On the other hand, we have no feeling for how hard the functionality proofs would be in general. A thoroughgoing comparison of the two approaches would be welcome.

Automating termination

Although there is very little work in this thesis on termination, our relationless function definitions were designed with the intent of supporting the methodical search for correct termination relations. The system reported in [38] seems to be the best current system for termination proofs; it uses inductively proved lemmas, systematic search for polynomial orderings, and a powerful decision procedure. It would be interesting to see how these ideas would work out in our setting.

More instantiations

ProofPower [7] and HOL-Light [50] are other ML implementations of the HOL logic, so they are prime candidates for TFL instantiations (even though HOL-Light is written in Caml-Light, a slightly different dialect of ML). LAMBDA is also a possible candidate, although it already has a powerful function definition facility based on a meta-theoretic model [36].

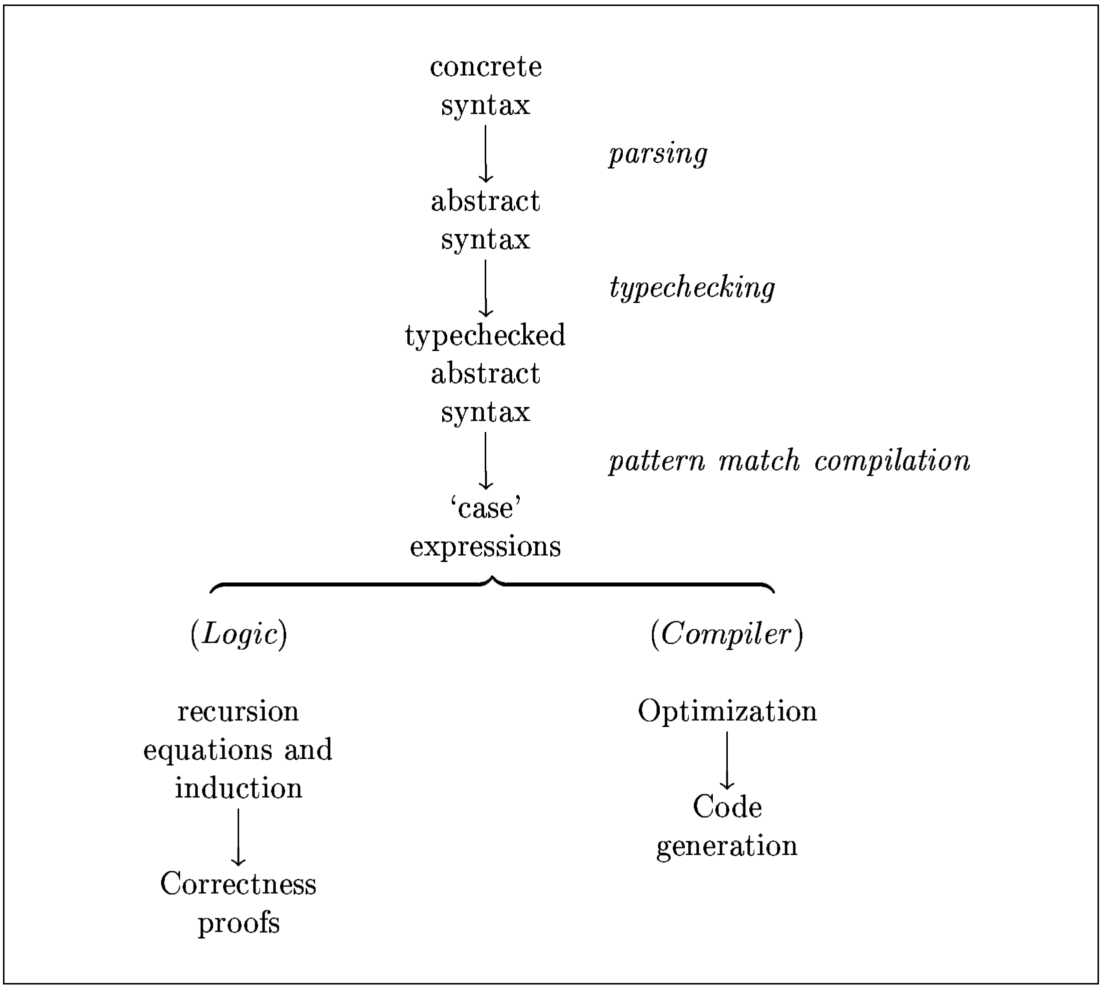
An interesting challenge is raised by set theory since, in contrast to HOL, it is essentially typeless; a good place to start would be the 'ZF' instantiation of Isabelle.

Strong induction

One should question whether induction theorems of the form that we derive are worth the trouble. After all, we are passing from strong induction (the well-founded induction theorem) to a (seemingly) weaker form: instead of assuming the property for *all* R-smaller terms, we are assuming it for only the recursive arguments in the body of the function. In many cases, the latter form suffices, but perhaps a high degree of automation could allow just the use of strong induction? This would simplify the situation, in that only the wellfounded induction theorem would be needed, and also potentially allow more proofs to go through. Some research on this has already been done in the context of proof planning [44, 58].

Integration with a compiler

Higher order logic theorem provers and functional language translators share a certain amount of similar functionality, then part ways. The following picture gives an indication of what we mean:



Both kinds of system parse into abstract syntax trees for sugared lambda calculus, then perform type inference. With TFL, the pattern matching compilation phase is also duplicated. However, things then diverge: on the logic side, the input recursion equations are derived as theorems, an induction theorem is proved, and further processing consists of proving various correctness properties involving the function. On the compiler side, waves of optimization take place before code is generated. It would be interesting to see whether the 'shared initial prefix' could be extended so that compiler optimization could benefit from proof. Some possibilities are the following:

- Much optimization is just simplification [10, 55], something that the theorem proving community has a great deal of experience with. In some cases, it seems that the wheel is being re-invented: for example, the program optimizer of [103] is built on the re-discovery—a decade and a half later—of Paulson's combinators for rewriting!

- It would be interesting to see if program transformations like the tail-recursion optimizations validated in Section 5.6 could be systematically applied in the optimization phase of a compiler. In fact Tullsen and Hudak[99] claim that program transformation of this type is much talked about, but little implemented. It may be that some theorem proving to eliminate applicability conditions and validate matches could allow these powerful optimizations to be brought to bear.

- Going further, it seems clear that powerful program optimizations like supercompilation [98] or Burstall and Darlington's transformation strategies [22] can be very easily implemented in a higher order logic proof system. Can they be applied by somehow using deduction inside the optimization stage of a compiler?

The research proposed here could be done by extending a theorem prover with a compiler middle-and-back end, or by embedding a higher order logic prover in the internals of an existing compiler. Related work is reported in [32]; that work differs by having been performed outside of any compiler.

A program development environment

What about using a higher order logic theorem prover as a medium for actually writing programs, lots of them? So-called logical program development is an idea that has been bandied about for various formalisms over many years [81, 31, 26], and it would be interesting to see how it could work in an HOL theorem prover.

First off, we should note the mismatch between functions in a logic and programs: yes, many functions can be expressed in HOL that are not computable; likewise, there are many programs that aren't total functions and so wouldn't fit accurately into HOL. However, these reservations are slightly beside the point that is important to us, which is that a vast number of interesting and useful algorithms can be directly defined and reasoned about in higher order logic. The problem is to see, for such algorithms, how a deductive environment aids or hinders the activity of programming. For example, putting correctness proofs temporarily aside, can programming be *more fun* in logic than in standard program development environments? That is, does the more formal environment offer new benefits?

We don't know a definitive answer to this; it's a matter for future work after all, but there may be some potential advantages for a programmer working in higher order logic:

- In classical logic, there is no built-in notion of function evaluation. Functions can be represented only by their defining equations, and notions such as evaluation strategy have no place. This is a wonderful simplification.
- One gains generality. The recursion equations of TFL offer a language in which to write abstract programs from which concrete programs in a wide variety of languages can be generated, essentially in a 'code generation' step. We do not expect this step to be at all trivial, since the gap between the logical context to the computational context is problematic for any number of reasons. However, this step can reasonably be thought of as being separate from the creation of the program.
- One gets free access to a library of mathematics which mingles smoothly with programs, since programs are just functions in the logic.
- One gets powerful algorithms used for theorem proving, *e.g.*, simplification. For example, it would seem that a useful debugger could be implemented by using conditional and contextual simplification. Similar work has recently been reported in the context of hardware verification [75].

A more subtle benefit of using a deductive framework is found in an article by Georg Kreisel[59]. We end with this:

> In the introduction to 'Was sind und was sollen ...' (form and function) Dedekind, very reasonably, asks in turn what function his foundation of arithmetic might have. His answer was, of course, rigour in the sense of 'proof without gaps'. This raises the further question of what use (tacitly, his idea(l) of) rigour might be, and Dedekind himself addressed it. His answer was "reliability". He illustrated it by reference to the familiar phenomena of 'reading', suggesting that genuine reliability requires spelling out words letter by letter! Taken literally, this is too far off the mark to reward elaboration; both concerning reading generally and reliability in particular; both reliability w.r.t. the written text and w.r.t. to the writer's thoughts that were to be recorded in the text. But the general idea – here the possibility of increasing reliability by formal representation, and thus in 'discrete bits' – is again demonstrably (by ENOD[1]), a real winner.

[1] Experience Not Only Doctrine.

Appendix A

System Architecture

In this appendix, we give an overview of the architecture of the TFL system, and describe some aspects of how it was instantiated to hol90 [96] and Isabelle/HOL [86]. Concretely, TFL is a programming language module parameterized with respect to a collection of modules that the client proof system must supply.[1]. A picture of the general structure of the system is given in Figure A.1.

A potential user of the services of TFL is in the following situation: if the client proof system can deliver suitable modules implementing the prelogic, supplying the right theorems, a definition principle, some rules of inference, and a few other bits and bobs, then TFL can be instantiated to supply a package for defining and reasoning about total recursive functions. Once this package has been instantiated, however, other facilities of the client proof system can be brought into play in order to get a more powerful tool for end users. In particular, there is a post-instantiation customization step, where automated provers for wellfoundedness and termination can be added as post-processors to the definition algorithms.

Thus TFL is—in its current incarnation—a package that a *developer* of a mechanized logic system would take and instantiate to his or her system: it is not a tool that an average user (someone mainly concerned with doing proofs) can currently take and use without some effort.

We will discuss the requirements for the client proof system in a bottom-up fashion, in general terms, before examining only a few details of the instantiations. In the following, reference will be made to various types being supplied by various modules: these types will be metalanguage types, not HOL types.

A.1 Requirements

The TFL system has of course been coded in a programming language (Standard ML [74]), and thus currently has the *pragmatic* limitation that the client proof

[1] Since TFL is implemented in Standard ML, a module is an ML *structure*, and a parameterized module is called a *functor*.

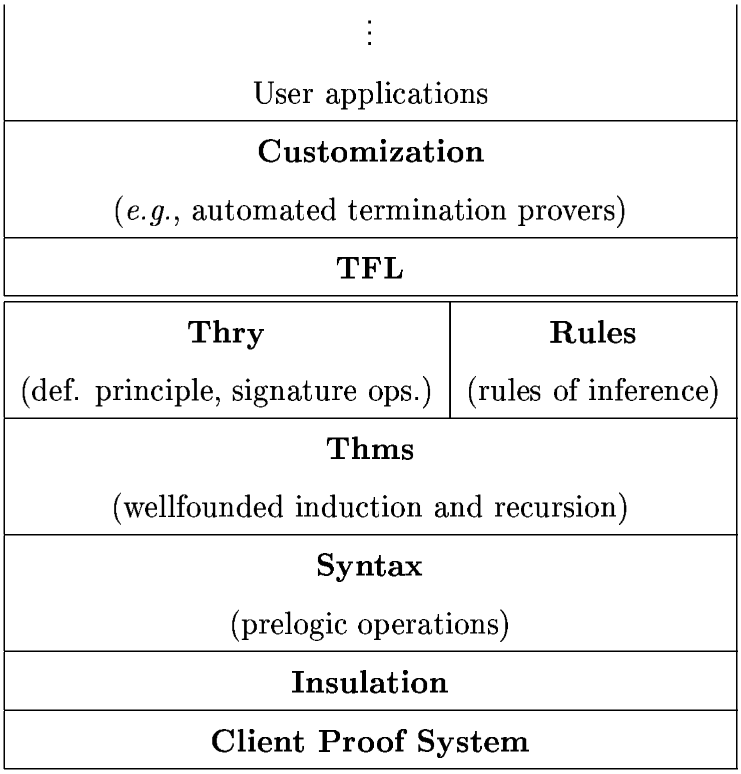


Figure A.1: System Structure

system must likewise be coded in Standard ML. However, component-based programming [45, 90] is becoming more well-supported, so it may soon be possible for an interlingual version of TFL to exist.

The second major requirement, without which no further progress can be made, is that the client proof system must implement the HOL logic, as described in Section 2.1.[2] Currently, some algorithms in TFL depend at crucial points on the use of classical logic, and the fact that the logic has an expressive type system is deeply rooted in the code, so users of constructive logic or set theory are currently unable to use TFL.

A.1.1 Insulation

To start, there is an *insulation layer*, which implements a small library of support routines for common facilities such as list processing, exception handling, and printing messages. This provides a standard base upon which the client is expected to provide the facilities in higher modules. Thus, this layer is *not*

[2] Or at least give a convincing imitation of such: TFL can't check that it is being instantiated to an implementation of HOL; however, in order for the algorithms of TFL to work correctly, the client system will have to do a very good job of impersonating higher order logic!

supplied by the client, but comes with the TFL package. The insulation layer enables, for example, a 'client-side' module to raise an exception that functions in the higher-level TFL module know how to interpret.

A.1.2 Syntax

The **Syntax** module deals with *abstract* syntax, not *concrete* syntax, and implements the prelogic as alluded to in Section 2.1. Metalanguage types for HOL types (holtype) and terms (preterm and term) are declared here, along with their associated operations. The preterm type is a type of lambda calculus terms that aren't required to be HOL terms, *i.e.*, a preterm need not be well-typed. Many useful syntactic operations can already be defined at the level of preterm. Going beyond that is the type term, which represents HOL terms, *i.e.*, preterms that are wellformed with respect to a given signature.

The functions in **Syntax** are used extensively in the TFL module for analyzing the structure of proposed definitions and building induction theorems. Basic prelogic operations are adequate for much of this: however, sometimes representation choices already made in the client proof system need to be given a uniform interface by Syntax.

For example, two common representations of relations in HOL are $(\alpha \times \alpha)$ set and $\alpha \rightarrow \alpha \rightarrow$ bool. The machinery in the TFL module needs to uniformly break elements of relations down, but we don't wish to mandate a particular representation for relations since that would limit the applicability of TFL. Thus Syntax is required to supply a function dest_relation that breaks apart a proposition stating inclusion in a relation, and returns the relation plus the pair of elements in the relation. For example, in a client system that implemented relations as $(\alpha \times \alpha)$ set, invoking dest_relation on the object-language term '$(x, y) \in R$' yields the (metalanguage) triple (R, x, y). Similarly, in a '$\alpha \rightarrow \alpha \rightarrow$ bool' client, dest_relation $(R\ x\ y)$ should also return (R, x, y).

A.1.3 Thms

The **Thms** module provides a type thm, which represents HOL theorems. The following three important theorems must be provided by **Thms**:

Wellfounded induction	$\mathsf{WF}(R)$ $\supset (\forall x.\ (\forall y.\ R\ y\ x \supset P\ y) \supset P\ x)$ $\supset \forall x.\ P\ x$
Restriction rewrite	$R\ x\ y \supset (f \,\vert\, R, y)\ x = f\ x$
Wellfounded recursion	$(f = \mathsf{WFREC}\ R\ M)$ $\supset \mathsf{WF}(R)$ $\supset \forall x.\ f(x) = M\ \ (f \,\vert\, R, x)\ x$

The exact syntactic form of these theorems is crucial: *e.g.*, supplying an equivalent, but not identical, wellfounded induction theorem will cause the algorithm for producing customized induction theorems of Section 3.4 to fail.

A.1.4 Thry

The **Thry** structure declares a type `theory`, which is a metalanguage collection of objects related to a signature Σ_Ω. **Thry** is required to supply the following services:

- Matching algorithms for `holtype` and `term`.

- A function that maps `preterms` to `terms` in Σ_Ω, *e.g.*, a type checking function.

- An implementation of the definition principle of Section 2.1.2. Once a definition is made, it is stored in the theory.

- A database of datatype facts of the kind listed in Section 2.2. The facts of crucial importance are the definition of the 'case' construct (used in the pattern-matching translation), the exhaustion theorem (used in proving pattern completeness for induction theorems), and the congruence theorem for the type (used to form contexts in termination condition extraction). It is convenient, but not necessary, if this database is automatically augmented when a new (object language) datatype is defined.

  Along with the congruence theorems arising from types, the user may also manually add congruence theorems for other constructs, *e.g.*, to handle instances of higher order recursion, the `let` construct, implication, bounded quantification, *etc.*

A.1.5 Rules

The **Rules** module supplies rules of inference, most of which can be found in Figure 2.1. Although the rules have quite simple specifications, the implementation of inference rules in client systems has been the most important design choice in the construction of TFL. We will discuss this more thoroughly in the following section.

Along with these basic inference rules, **Rules** must also supply a 'heavy-weight' rule of inference: the contextual rewriter used for termination condition extraction, as described in Section 2.7.

A.2 Instantiations

The TFL package has been instantiated to two different theorem provers: hol90 and Isabelle/HOL [86]. Although these are both written in same programming language and implement the same logic, the two theorem provers differ a great deal in the details. The essential difference between the two systems is that hol90 is a direct implementation of the HOL logic, while Isabelle/HOL is an instantiation of the Isabelle *logical framework* to the HOL logic.

In order to avoid being trapped in a thicket of system particulars in the following, we will discuss the two instantiations only from the point of view of **Thry** and **Rules**, since theories and rules of inference presented the largest differences that had to be overcome. By restricting the discussion, a wealth of detail will be passed over in silence.

A.2.1 Theories

Work in hol90 takes place against an implicit background theory known as the *current theory*. New definitions are added to the current theory by updating a reference cell; thus, hol90 theories are *imperative* in nature. In contrast, a theory in Isabelle/HOL is *functional* in nature: adding a new definition to a theory results in a new and different theory being created; subsequently the new and old theory both exist, but are different. Since TFL alters the theory when asked to define a function, a solution implementable by both approaches was needed. The standard functional programming idiom of explicitly passing the state (in this case, an item of type `theory`) throughout the code was adopted. This directly suited the approach already taken in Isabelle. It is also easy to accommodate in hol90: the extra `theory` parameter is just ignored, since any accesses to the theory can be directly made by accessing the current theory.

A.2.2 Rules

In hol90, as in its ancestor LCF, inference rules are implemented as metalanguage functions. For example, *modus ponens* is implemented by approximately the following code:

```
fun MP th1 th2 =
  let val (hyps1, c1) = dest_thm th1
      val (hyps2, c2) = dest_thm th2
      val (antecedent,consequent) = dest_imp c1
  in
     if aconv antecedent c2
     then THM (union hyps1 hyps2, consequent)
     else Fail "modus ponens failed"
  end
```

(where `fun` is a keyword signifying that a function is being defined, `val` is a keyword signifying that a variable binding is being made, `dest_thm` breaks a theorem into a (hypotheses, conclusion) pair, `dest_imp` breaks an implication into an (antecedent, consequent) pair, `aconv` implements the test for α-convertibility, `union` joins two hypothesis lists together, `THM` constructs an element of the `thm` type, and `Fail` raises an exception.)

Isabelle

The Isabelle system [86] is a *logical framework* capable of implementing a wide variety of different logics. In Isabelle, a logic of interest—the *object logic*—is embedded in the Isabelle *meta-logic*. For example, the Isabelle/HOL system was obtained by embedding the HOL logic in the Isabelle meta-logic. The Isabelle meta-logic is a very simple lambda calculus-based higher order logic having only connectives for equality (==), implication ($\supset$), and universal quantification (!!).

$$
\begin{array}{llll}
\supset\text{-intro} & \dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \supset Q} & \dfrac{\Gamma \vdash P \supset Q \quad \Delta \vdash P}{\Gamma \cup \Delta \vdash Q} & \supset\text{-elim} \\[2em]
\text{!!-intro} & \dfrac{\Gamma \vdash P}{\Gamma \vdash !!x.\ P} & \dfrac{\Gamma \vdash !!x.\ P}{\Gamma \vdash [x \mapsto N]P} & \text{!!-elim}
\end{array}
$$

Figure A.2: Isabelle meta-logic

The basic rules are given in Figure A.2 (the equality rules are omitted, and the introduction rule for universal quantification has the usual proviso that x not be free in Γ). The rules of the Isabelle meta-logic are implemented as metalanguage programs.

Isabelle represents object-logic rules with assertions of the general form

$$[\phi_1, \ldots, \phi_m] \Rightarrow \phi.$$

This is just a layer of syntax that translates to

$$\vdash \phi_1 \supset \ldots \supset \phi_m \supset \phi.$$

Several rules that could be derived are also coded in the metalanguage, chief among them being a higher order resolution rule:

$$\boxed{\quad \frac{[\psi_1, \ldots, \psi_m] \Rightarrow \psi \qquad [\phi_1, \ldots, \phi_n] \Rightarrow \phi}{\sigma([\phi_1, \ldots, \phi_{i-1}, \psi_1, \ldots, \psi_m, \phi_{i+1}, \ldots, \phi_n] \Rightarrow \phi)} \quad \sigma(\psi) = \sigma(\phi_i). \quad}$$

Now meta-logic inference can be used to implement object logic inference, since an object logic proof step can be achieved by an application of the resolution rule. Thus the *modus ponens* rule of inference (using $\rightarrow$ for the *object* level implication connective) can be rendered in Isabelle as

$$[P \rightarrow Q, P] \Rightarrow Q.$$

This rendering obscures the real structure, which uses the Trueprop constant to separate the derivability judgement of the object logic from that of the meta-logic. Thus the underlying representation is the following:

$$\vdash \mathsf{Trueprop}(P \rightarrow Q) \supset \mathsf{Trueprop}(P) \supset \mathsf{Trueprop}(Q). \tag{A.1}$$

The Isabelle/HOL instantiation maintains a very close association between the meta-logic connectives and their counterparts in the object-logic. Thus we had to think hard about which connectives to choose for certain operations, particularly since so much of the work of TFL amounts to applications of equality, implication, and universal quantification.

Which rules?

Now that we have discussed the implementation of inference rules in both systems, we can justify our choice. A choice *had* to be made, since TFL must operate uniformly, no matter what the underlying proof system. Roughly, the choice was between rules as programs, or as theorems. The author chose rules-as-programs, largely because it seemed to lead to less total effort in instantiating both systems. For example, having rules-as-theorems seemed to imply that the higher order resolution rule would have to be applied directly inside TFL, which would require an implementation of higher order resolution to be supplied by any client, hol90 in particular, with the further requirement of having to behave identically to the Isabelle resolution rule.

Daunted by this prospect, we chose rules-as-programs. However, that was only the interface with the upper TFL layer; in the Isabelle/HOL instantiation, we used resolution as much as possible. For example, the actual implementation of *modus ponens* that we used was the following (where RS is an infix identifer denoting the resolution rule and the identifier mp denotes (A.1)):

```
fun MP th1 th2 = th2 RS (th1 RS mp);
```

So in general, in the Isabelle/HOL implementation of **Rules**, we tried to stay at the object logic level as much as possible and use resolution as much as possible. However, in some cases, the meta-logic needed to be used, or was simply more convenient. A case in point is the extraction of termination conditions (Section 3.2). Recall that the small proof performed to capture termination conditions was invoked as the 'condition solver' for the contextual rewriter. The essential step in the proof is that the termination conditions get *assumed* and stored away on the assumptions. Making assumptions, and discharging them, is a facility that only exists at the meta-logic level.

A.3 Customization

The following is a simple example of the postprocessors that can be applied to the results of definitions in order to prove wellfoundedness and eliminate any essentially trivial termination conditions:

- The wellfoundedness prover just backchains on theorems that propagate wellfoundedness. A standard collection of these is the following:

$$\begin{array}{rcl} & & \vdash \mathsf{WF}\ (<) \\ & & \vdash \mathsf{WF}\ \mathsf{pred} \\ & & \vdash \mathsf{WF}\ (\mathsf{measure}\ f) \\ \vdash \mathsf{WF}(R) \wedge \mathsf{WF}(Q) & \supset & \vdash \mathsf{WF}\ (\mathsf{LEX}\ R\ Q) \\ \vdash \mathsf{WF}(R) & \supset & \mathsf{WF}(\mathsf{inv\_image}\ R\ f) \\ \vdash \mathsf{WF}(R) & \supset & \mathsf{WF}(\mathsf{TC}\ R) \end{array}$$

- The termination prover rewrites with the following definitions:

$$\begin{array}{rcl} \mathsf{measure} & \equiv & \mathsf{inv\_image}\ (<) \\ \mathsf{inv\_image}\ R\ f & \equiv & \boldsymbol{\lambda} x\ y.\ R\ (f\ x)\ (f\ y) \\ \mathsf{LEX}\ R_1\ R_2\ (u,v)\ (x,w) & \equiv & R_1\ u\ x \vee (u = x \wedge R_2\ v\ w) \end{array}$$

  and then invokes a linear arithmetic package.

Example. The gcd function

$$\begin{array}{rcl}
\mathsf{gcd}(0, y) & \equiv & y \\
\mathsf{gcd}(\mathsf{Suc}\ x, 0) & \equiv & \mathsf{Suc}\ x \\
\mathsf{gcd}(\mathsf{Suc}\ x, \mathsf{Suc}\ y) & \equiv & \ \texttt{if}\ y \leq x\ \texttt{then}\ \mathsf{gcd}(x - y, \mathsf{Suc}\ y) \\
& & \ \texttt{else}\ \mathsf{gcd}(\mathsf{Suc}\ x, y - x)
\end{array}$$

when defined by supplying the termination relation

$$\mathsf{measure}\ (\mathsf{pair\_case}\ +),$$

which is a terse way to say that the sum of the sizes of the arguments to gcd decreases, gives rise to the following termination conditions:

$$\begin{array}{r}
\mathsf{WF}\ (\mathsf{measure}\ (\mathsf{pair\_case}{+})), \\
(\neg y \leq x) \supset (\mathsf{Suc}\ x + (y - x)) < (\mathsf{Suc}\ x + \mathsf{Suc}\ y), \\
y \leq x \supset ((x - y) + \mathsf{Suc}\ y) < (\mathsf{Suc}\ x + \mathsf{Suc}\ y).
\end{array}$$

The wellfoundedness prover eliminates the first constraint in one step. The other termination conditions are in the range of the linear arithmetic prover.
□

Future work should focus on extending this functionality to search for correct termination relations for definitions given in the relationless style.

Bibliography

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects.* Springer-Verlag, New York, 1996.

[2] S. Agerholm. LCF examples in HOL. *The Computer Journal*, 38(2):121–130, July 1995.

[3] S. Agerholm. Non-primitive recursive function definition. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications (LNCS 971)*, pages 17–31, Aspen Grove, Utah, September 1995. Springer-Verlag.

[4] S.F. Allen, R. Constable, D. Howe, and W. Aitken. The semantics of reflected proof. In *Fifth annual IEEE symposium on Logic in Computer Science*, pages 95–107, Philadelphia, USA, June 1990.

[5] Penny Anderson and David Basin. Program development schemata as derived rules. *Journal of Symbolic Computation*, 2000. To appear.

[6] Peter Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.

[7] R. D. Arthan. A report on ICL HOL. In Phillip Windley, Myla Archer, Karl Levitt, and Jeffrey Joyce, editors, *International Tutorial and Workshop on the HOL theorem proving system and it Applications*, University of California at Davis, August 1991. ACM-SIGDA / IEEE Computer Society, IEEE Computer Society Press.

[8] Lennart Augustsson. Compiling pattern matching. In J.P. Jouannnaud, editor, *Conference on Functional Programming Languages and Computer Architecture (LNCS 201)*, pages 368–381, Nancy, France, 1985.

[9] Franz Baader and Tobias Nipkow. *Term Rewriting and all that.* Cambridge University Press, 1998.

[10] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming*, Blatimore, Maryland, September 1998. ACM Press.

[11] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL - lessons learned in Formal-Logic Engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'99)*, number 1690 in LNCS, Nice, 1999. Springer-Verlag.

[12] Richard Bird. Functional algorithm design. In B. Moeller, editor, *Mathematics of Program Construction, Third International Conference, (MPC'95)*, volume LNCS 947, pages 2–17, Kloster Irsee, Germany, July 17-21 1995.

[13] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

[14] Barry Boehm. Software engineering. *IEEE Transactions on Computers*, C-25(12):1226–1241, 1976.

[15] Adel Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23(1):47–77, 1997.

[16] Richard Boulton. *Efficiency in a Fully-Expansive Theorem Prover*. PhD thesis, University of Cambridge, May 1994. Technical Report Number 337, University of Cambridge Computer Laboratory.

[17] Richard Boulton. Multi-predicate induction schemes for mutual recursion. To appear as technical report in Division of Informatics, University of Edinburgh, 2000.

[18] Ana Bove. Programming in Martin-Löf Type Theory. Unification: A non-trivial Example. Master's thesis, Chalmers University of Technology, 1999. Licentiate Thesis.

[19] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, 1979.

[20] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[21] Rod Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, February 1969.

[22] Rod Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[23] Alonzo Church. A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[24] Martin D. Coen. *Interactive Program Derivation.* PhD thesis, University of Cambridge, November 1992. Technical Report Number 272, University of Cambridge Computer Laboratory.

[25] Avra Cohn and Robin Milner. On using Edinburgh LCF to prove the correctness of a parsing algorithm. Technical Report CSR-113-82, Dept. of Computer Science, Edinburgh University, 1982.

[26] Robert Constable, S. Allen, H. Bromly, W. Cleaveland, J. Cremer, R. Harper, D. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. Sasaki, and S. Smith. *Implementing Mathematics With the Nuprl Proof Development System.* Prentice-Hall, New Jersey, 1986.

[27] C. Cornes. *Conception d'un Langage de Haut Niveau de Representation de Preuves: recurrence par filtrage de motifs, unification en presence de types inductif primitifs. synthese de lemmes d'inversion.* These d'universite, Paris 7, November 1997.

[28] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North-Holland, 1958. Two sections by William Craig.

[29] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag. Math.*, 34:381–392, 1972.

[30] Richard Dedekind. *Essays on the theory of numbers.* Dover, 1963. Authorized translations by Wooster W. Beman of *Stetigkeit und irrationale Zahlen* and *Was sind und sollen die Zahlen* for Open Court Publishing, 1901.

[31] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[32] A. Dold, F. W. von Henke, H. Pfeifer, and H. Ruess. Formal verification of transformations for peephole optimization. In *Proceedings of FME'97 (LNCS 1313)*, pages 459–472. Springer-Verlag, September 1997.

[33] Axel Dold. Representing, verifying and applying software development steps using the PVS system. In V.S. Alagar and Maurice Nivat, editors,

Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, AMAST'95, Montreal, volume 936 of *Lecture Notes in Computer Science*, pages 431–435. Springer-Verlag, 1995.

[34] Axel Dold. Software development in PVS using generic development steps. To appear in Springer LNCS, Proceedings of a Seminar on Generic Programming, April 1998.

[35] William Farmer. Recursive definitions in IMPS. Available by anonymous FTP at `ftp.harvard.edu`, in directory `imps/doc`, file name `recursive-definitions.dvi.gz`, 1997.

[36] Simon Finn, Mike Fourman, and John Longley. Partial functions in a total setting. *Journal of Automated Reasoning*, 18(1):85–104, February 1997.

[37] P. Flener, K.-K. Lau, and M. Ornaghi. On correct program schemas. In N.E. Fuchs, editor, *Proceedings of LOPSTR'97 (LNCS 1463)*, pages 124–143. Springer-Verlag, 1998.

[38] Jürgen Giesl. *Automatisierung von Terminieringsbeweisen für rekursiv defininierte Algorithmen*. PhD thesis, Technische Hochshule Darmstadt, 1995.

[39] Jürgen Giesl. Termination analysis for functional programs using term orderings. In *Proceedings of the 2nd International Static Analysis Symposium*, Glasgow, Scotland, 1995. Springer-Verlag.

[40] Jürgen Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19(1):1–29, August 1997.

[41] Jean-Yves Girard. *Proof Theory and Logical Complexity*, volume 1. Bibliopolis, 1987.

[42] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[43] Mike Gordon and Tom Melham. *Introduction to HOL, a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[44] Jeremy Gow, Alan Bundy, and Ian Green. Extensions to the estimation calculus. Technical Report RP953, Edinburgh University Department of Artificial Intelligence, 1999.

[45] The Object Management Group. Common Object Request Broker Architecture and Specification. Technical report, The Object Management Group, OMG Headquarters, 492 Old Connecticut Path, Framingham, Massachusetts 01701, USA, February 1998. Also at http://www.om.org.

[46] E. L. Gunter. A broader class of trees for recursive type definitions for HOL. In J. J. Joyce and C.-J. H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop (HUG'93)*, number 780 in Lecture Notes in Computer Science, pages 141–154, Vancouver, B.C., August 11-13 1994. Springer-Verlag.

[47] H. Busch. Unification based induction. In L.J.M. Claesen and M.J.C. Gordon, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 97–116, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, North-Holland. IFIP Transactions.

[48] John Harrison. Inductive definitions: automation and application. In E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors, *Proceedings of the 1995 International Workshop on Higher Order Logic theorem proving and its applications*, number 971 in LNCS, pages 200–213, Aspen Grove, Utah, 1995. Springer-Verlag.

[49] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre, 1995.

[50] John Harrison. HOL-Light: A tutorial introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume LNCS 1166, pages 265–269. Springer-Verlag, 1996.

[51] John Harrison. *Theorem Proving with the Real Numbers.* CPHC/BCS Distinguished Dissertations. Springer, 1998.

[52] P. V. Homeier and D. F. Martin. A verified verification condition generator. *The Computer Journal*, 38(2):131–141, July 1995.

[53] Gerard Huet and Bernhard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[54] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987. With contributions by P. Wadler and P. Hancock.

[55] Simon Peyton Jones and Andre Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, September 1998.

[56] Deepak Kapur and M. Subramaniam. Automating induction over mutually recursive functions. In *Proceedings of the 5th International Conference on*

Algebraic Methodology and Software Technology (AMAST'96), volume 1101 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[57] Delia Kesner, Laurence Puel, and Val Tannen. A typed pattern calculus. *Information and Computation*, 124(4):32–61, 1996.

[58] Ina Kraan, David Basin, and Alan Bundy. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16:113–145, 1996.

[59] Georg Kreisel. Logical aspects of computation: contributions and distractions. In P.G. Oddifreddi, editor, *Logic and Computer Science*. Academic Press, 1990. Number 31 in the APIC Studies in Data Processing Series.

[60] L. Claesen and M. Gordon, editors. *International Workshop on Higher Order Logic Theorem Proving and its Applications*, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, Elsevier Science Publishers.

[61] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[62] P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, 1966.

[63] Jeffery R. Lewis, Mark B. Shields, Erik Meijer, and John Launchbury. Implicit parameters: Dynamic scoping with static types. In Tom Reps, editor, *ACM Symposium on Principles of Programming Languages*, Boston, Massachusetss, USA, January 2000. ACM Press.

[64] Lena Magnusson and Bengt Nordstrom. The ALF proof editor and its proof engine. In *Types for Proofs and Programs (LNCS 806)*, pages 213–237, Nijmegen, Netherlands, 1994. Springer-Verlag.

[65] Zohar Manna and Richard Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48, 1981.

[66] Pascal Manoury. A user's friendly syntax to define recursive functions as typed λ-terms. In *Types for Proofs and Programs: International Workshop TYPES'94*, number 996 in Lecture Notes in Computer Science, Baastad, Sweden, June 1995. Springer-Verlag.

[67] Luc Maranget. Two techniques for compiling lazy pattern matching. Technical Report 2385, INRIA, October 1994.

[68] Per Martin-Löf. Constructive mathematics and computer programming. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, Prentice-Hall International Series in Computer Science, pages 167–184. Prentice-Hall, 1985.

[69] Conor McBride. *Dependently Typed Functional Programs and their Proofs.* PhD thesis, University of Edinburgh, 1999.

[70] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* MIT Press, 2 edition, 1965.

[71] Tom Melham. Automating recursive type definitions in higher order logic. In Graham Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[72] Tom Melham. A package for inductive relation definitions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 350–357. IEEE Computer Society Press, Davis, California, USA, August 1991.

[73] Tom Melham. The HOL logic extended with quantification over type variables. In L. Claesen and M. Gordon [60].

[74] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[75] J Moore. Symbolic simulation: An ACL2 approach. In G. Gopalakrishnan and P. Windley, editors, *Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD'98)*, volume LNCS 1522, pages 334–350. Springer-Verlag, November 1998.

[76] Olaf Müller. *A verification environment for I/O automata based on formalized meta-theory.* PhD thesis, Institut für Informatik, Technische Universität München, September 1998.

[77] Tobias Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[78] Bengt Nordstrom. Terminating general recursion. *BIT*, 28:605–619, 1988.

[79] S. Owre, J. M. Rushby, N. Shankar, and D.J. Stringer-Calvert. *PVS System Guide.* SRI Computer Science Laboratory, September 1998. Available at `http://pvs.csl.sri.com/manuals.html`.

[80] C. Parent. Synthesizing proofs from programs in the calculus of inductive constructions. In *Third International Conference on the Mathematics of Program Construction*, number 947 in Lecture Notes in Computer Science, pages 351–379. Springer-Verlag, July 1995.

[81] Helmut A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development.* Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[82] Lawrence Paulson. A higher order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[83] Lawrence Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 3:143–170, 1985.

[84] Lawrence Paulson. Proving termination of normalization functions for conditional expressions. *Journal of Automated Reasoning*, 2:63–74, 1986.

[85] Lawrence Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *12th International Conference. on Automated Deduction (CADE)*, volume LNAI 814, pages 148–161. Springer-Verlag, 1994. Revised version available at `http://www.cl.cam.ac.uk/users/lcp/papers/recur.html` under title 'A Fixedpoint Approach to (Co)inductive and Co(datatype) Definitions'.

[86] Lawrence Paulson. *Isabelle : A Generic Theorem Prover.* Number 828 in LNCS. Springer-Verlag, 1994. Up-to-date reference manual can be found at `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/`.

[87] Lawrence Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7:175–204, March 1997.

[88] Henrik Persson. *Type Theory and the Integrated Logic of Programs.* PhD thesis, Chalmers University of Technology, June 1999.

[89] Franz Regensburger. *HOLCF: Eine konservative Einbettung von LCF in HOL.* PhD thesis, Institut für Informatik, Technische Universität München, 1994.

[90] D. Rogerson. *Inside COM.* Microsoft Press, 1996.

[91] Joseph Rouyer. Développement d'algorithme d'unification dans le Calcul des Constructions avecs types inductifs. Technical Report 1795, INRIA-Lorraine, November 1992.

[92] Piotr Rudnicki and Andrzej Trybulec. On equivalents of well-foundedness. *Journal of Automated Reasoning*, 23(3):197–234, 1999.

[93] H. Schwichtenberg and S. Wainer. Ordinal bounds for programs. In Jeff Remmel, editor, *Feasible Mathematics II*, pages 387–406. Birkhäuser, 1994.

[94] Dana Scott. A type theoretic alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1-2):411–440, December 1993.

[95] Natarajan Shankar. Steps towards mechanizing program transformations using PVS. In B. Moeller, editor, *Mathematics of Program Construction, Third International Conference, (MPC'95)*, number 947 in Lecture Notes in Computer Science, pages 50–66, Kloster Irsee, Germany, July 17-21 1995.

[96] Konrad Slind. An implementation of higher order logic. Technical Report 91-419-03, University of Calgary Computer Science Department, 1991. Masters Thesis.

[97] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[98] M.H. Sorenson, R. Glück, and N.D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, November 1996.

[99] Mark Tullsen and Paul Hudak. Shifting expression procedures into reverse. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical report BRICS-NS-99-1, University of Aarhus, pages 95–104, San Antonio, Texas, January 1999.

[100] D.A. Turner. Miranda: a non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architectures (FPCA)*, number 201 in Lecture Notes in Computer Science, pages 1–16. Springer-Verlag, 1985.

[101] David A. Turner. Elementary strong functional programming. In Pieter H. Hartel and Marinus J. Plasmeijer, editors, *Functional Programming Languages in Education, First International Symposium, FPLE'95*, volume 1022 of *Lecture Notes in Computer Science*, pages 1–13, Nijmegen, The Netherlands, December 1995. Springer-Verlag.

[102] M. van der Voort. Introducing well-founded function definitions in HOL. In L. Claesen and M. Gordon [60].

[103] E. Visser, Z. el A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In P. Hudak and C. Queinnec, editors, *The 1998 International Conference on Functional Programming (ICFP'98)*, Baltimore, Maryland, September 1998. ACM.

[104] Christoph Walther. On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157, 1994.

[105] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 1(27):164–180, January 1980.

[106] Hao Wang. Towards mechanical mathematics. *IBM Journal*, 4(2–22), 1960.

[107] Jon Whittle. *The Use of Proofs-as-Programs to Build an Analogy-Based Functional Program Editor*. PhD thesis, Division of Informatics, University of Edinburgh, Scotland, 1999.

[108] Martin Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.