Institut für Informatik
der Technischen Universität München

Automating Recursive Definitions and Termination Proofs in Higher-Order Logic

Alexander Krauss

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

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Helmut Seidl
Prüfer der Dissertation:		
	1.	Univ.-Prof. Tobias Nipkow, Ph.D.
	2.	Prof. Lawrence C. Paulson, Ph.D. University of Cambridge, UK

Die Dissertation wurde am 29.01.2009 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 02.07.2009 angenommen.

Abstract

The aim of this thesis is to provide an infrastructure for general recursive function definitions in a proof assistant based on higher-order logic (HOL) that has no native support for recursion or pattern matching.

In the first part we develop a tool that automates recursive function definitions and provides appropriate proof rules for them. Compared to previous work, our package supports the definition of partial functions, modeling the domain of the function by an inductive domain predicate. An automatically-generated partial induction rule makes partial correctness proofs independent from termination proofs. This modularity considerably facilitates termination arguments for nested recursions.

The second part addresses the problem of automatically solving the termination proof obligations that arise from function definitions. Methods from the literature can be applied, but require significant adaptation to the specific needs of our setting: They must produce full formal proofs and work relative to a rich interactive theory. Our approach encompasses a rule-based selection of measure functions, a simple control-flow analysis inspired by the dependency-pairs approach, and a modified version of the size-change principle based on certificates. A formalization of the full size-change principle is also provided.

In the third part we discuss how pattern matching, which occurs frequently in functional programming, can be supported in HOL function definitions. We present a very general form of pattern matching, where arbitrary expressions can serve as patterns. We show how such patterns can be encoded using a custom matching combinator and how their consistency can be expressed in proof obligations.

We also study the problem of transforming ML-style sequential pattern matching into minimal sets of independent equations, such that they are consistent in HOL. We relate the problem to the minimization problem for propositional DNF formulas and show that it is Σ_2^P-complete. We then develop a concrete algorithm that computes minimal patterns.

As another application of the new set of tools, we show how user-specified induction schemes can be generated from simpler properties, which often makes their proofs fully automatic.

Zusammenfassung

Ziel dieser Arbeit ist die Entwicklung einer Infrastruktur für rekursive Funktionsdefinitionen in einem interaktiven Theorembeweiser, in dem Rekursion und Pattern-Matching nicht von Haus aus unterstützt werden. Wir arbeiten im Kontext höherstufiger Logik (higher-order logic, HOL).

Im ersten Teil entwickeln wir ein Werkzeug, welches Funktionsdefinitionen automatisiert und geeignete Beweisregeln dafür bereitstellt. Im Gegensatz zu existierenden Ansätzen unterstützt unser Verfahren auch partielle Funktionen, die mit Hilfe eines induktiven Domainprädikats modelliert werden. Eine automatisch generierte Induktionsregel erlaubt partielle Korrektheitsbeweise unabhängig von der Terminierung der Funktion. Diese modulare Struktur erleichtert insbesondere die Behandlung von geschachtelter Rekursion.

Der zweite Teil behandelt automatische Terminierungsbeweise, um die aus Funktionsdefinitionen entstehenden Beweisverpflichtungen automatisch zu lösen. Dabei verwenden wir Methoden aus der Literatur, die allerdings an die spezifischen Anforderungen unseres Szenarios angepasst werden müssen. Unser Ansatz beinhaltet eine regelbasierte Auswahl von Maßfunktionen, eine einfache Kontrollflussanalyse ähnlich der Dependency-Pairs-Methode, und eine Variante des Size-Change-Kriteriums mit Zertifikaten. Eine Formalisierung des vollen Size-Change-Kriteriums wird ebenfalls entwickelt.

Im dritten Teil untersuchen wir, wie das in der funktionalen Programmierung gebräuchliche Pattern-Matching in HOL unterstützt werden kann. Wir entwickeln eine sehr allgemeine Form von Pattern-Matching in der Logik, welches beliebige Terme als Patterns erlaubt und mit Hilfe eines speziellen Kombinators in der Logik ausgedrückt werden kann. Die Konsistenz der Spezifikation wird durch spezielle Beweisverpflichtungen sichergestellt.

Außerdem untersuchen wir das Problem, Spezifikationen mit sequenziellem Pattern-Matching in reine Gleichungsspezifikationen mit möglichst wenigen Gleichungen umzuwandeln. Wir ziehen eine Parallele zum Problem der Minimierung Boolescher Ausdrücke in DNF und zeigen, dass das Minimierungsproblem für Patterns Σ_2^P-vollständig ist. Wir geben auch einen konkreten Algorithmus zur Minimierung von Patterns an.

Als weitere Anwendung der neuen Werkzeuge zeigen wir, wie sich vom Benutzer vorgegebene Induktionsschemata automatisch auf einfachere Beweisziele reduzieren lassen, wodurch sich der Beweis solcher Schemata oft weitgehend automatisieren lässt.

To Katharina — with nonterminating love

Acknowledgements

I am deeply indebted to Tobias Nipkow for offering me the opportunity to work with him, and for being such a pleasant supervisor in every respect. He greatly contributes to my education not only in logic and computer science, but also in other areas, such as contemporary music. Working under his guidance is great fun.

I am also grateful to Larry Paulson for agreeing to act as a referee and examiner, and for creating the next 700 theorem provers.

I want to thank all present and former members of the Isabelle group at TUM for making it such a fertile research biotope and such a friendly place. They all influenced this work in one or the other way: Clemens Ballarin, Stefan Berghofer, Jasmin C. Blanchette, Sascha Böhme, Amine Chaieb, Florian Haftmann, Julien Narboux, Steven Obua, Norbert Schirmer, Christian Urban, Tjark Weber, Makarius M. Wenzel, and Martin Wildmoser.

Jasmin and Makarius deserve special thanks for reading parts of this thesis and decorating it with countless colourful annotations and corrections. Florian, who shared the office with me in the beginning, has become something like a travel companion on this journey (and various sub-journeys). I always enjoy being exposed to his detailed knowledge of Bavarian history and his bizarre humour — often both at the same time. Thanks are also due to Lukas Bulwahn and Armin Heller, who worked on the implementation of the termination provers described in Chapter 3. They did a very good job.

Finally I would like to thank the numerous users of Isabelle, without whom much of this work would be meaningless.

My wife, Katharina Kühn, provided moral support of all kinds during the whole time, and especially in the end. I cannot thank her enough.

Parts of this research were financially supported by DFG Project Ni 491/10-1.

Contents

Chapter 1

Introduction

Contents

The purpose of this thesis is to provide better infrastructure for the definition of recursive functions in an interactive theorem prover. Such an infrastructure is essential for the practical usability of a proof assistant, since definitions can become very complex in nontrivial formalizations. However, recursive specifications can become inconsistent if the function is not terminating or otherwise ill-defined.

In traditional mathematical texts, these issues are typically not treated formally, and it is left to the the reader's intuition and experience to check that the definitions are well-formed. When working formally with an interactive theorem prover, such omissions are inacceptable. However, doing all the tedious justifications by hand for every function definition is not an attractive option either, and would severely limit the practical usability of the prover.

As a solution, we provide a tool which soundly introduces complex recursive definitions automatically by reducing them to first principles. This approach has been taken before [98], but our particular method solves several technical problems that have made working with general recursive functions hard in the past.

1.1 Contributions

In his PhD thesis [98], Konrad Slind describes a definitional specification mechanism for total recursive functions based on a wellfounded recursion operator. The package is called TFL (Terminating Functional Language) and is implemented for the theorem provers HOL4 [44] and Isabelle/HOL [81]. Developed about ten years ago, it still marks the state of the art for function definition packages in theorem provers, and the specific contributions of the present work are best understood in comparison with TFL. We discuss other related work in the individual chapters.

Partiality While TFL supports the definition of total functions only, we show how to define partial functions from recursive specifications that may be nonterminating. Such partial functions are modelled as total functions together with a domain predicate. A special induction rule allows reasoning about the functions in such a way that partial correctness proofs are orthogonal to termination proofs.

Nested Recursion As a pleasant side effect of partiality, our approach deals nicely with nested recursive functions, which have posed serious difficulties in all previous approaches, since their termination proof depends on the proof of some partial correctness property. Since termination and partial correctness are cleanly separated, all problems with nested recursion disappear.

Pattern matching Compared to TFL, which compiles pattern matching to nested case expressions, we propose a more flexible and expressive notion of pattern matching, where patterns are not syntactically restricted to datatype constructors. This supports a wider class of definitions and solves some issues with overlapping patterns.

We also study pattern transformations that are necessary to transform sequential pattern matching to purely equational specifications definable in HOL, which can sometimes lead to a large blowup in the number of equations. To avoid such an explosion where possible, we study the underlying minimization problem. However, the results are negative: It turns out that the explosion is unavoidable in some cases, and finding a minimal set of patterns is computationally expensive — the problem is Σ_2^P-complete. We also give a concrete algorithm that can minimize patterns.

Automated termination proofs We develop automated methods to solve the termination proof obligations that arise from function definitions. TFL provides no automation here, and requires the user to provide termination arguments manually.

Existing approaches for termination proofs of programs (in different formalism, e.g., term rewrite systems) must be adapted to be applicable in our particular scenario. Our modular approach includes a rule-based selection of measure functions, a simple control-flow analysis inspired by the popular dependency-pairs approach [4], and a modified version of the size-change principle [66] based on certificates. To our knowledge, this is the first procedure that proves termination of a nontrivial class of recursive functions in an LCF-style theorem prover.

User-specified induction rules We show how our tools can also be used to derive induction rules that are given by the user instead of generated by a package. This often drastically simplifies the proofs of custom induction rules.

1.2 Interactive Theorem Proving

From the nature of formal proofs, it is apparent that the help of computers is necessary to develop, check and maintain them, once the development has reached a certain size.

Interactive theorem proving is based on the idea that humans and computers can best combine their respective strengths when they interact. The role of the human is to guide the overall proof and formalize the intuitive ideas. The role of the machine is to check the correctness of the proof under development. Additionally, the system provides assistance of various kinds: different forms of proof automation, decision procedures, presentation, counterexample generation, type inference and many more.

Systems that support this mode of interaction are called *interactive theorem provers* or *proof assistants*. Among the most widely known systems in this category are ACL2 [60], Coq [15], HOL4 [44], HOL Light [51], Mizar [79], PVS [85] and Isabelle [90, 81], on which this work is based.

LCF approach It is an important design principle to keep the different activities of *producing* and *checking* formal proofs clearly separated. The LCF approach, which first appeared in the Edinburgh LCF system [45] and is used in most of the aforementioned systems, achieves this in a simple and effective way: Proved theorems are implemented as an abstract datatype *thm*, and the type system of the implementation language (in Isabelle: Standard ML [74]) ensures that values of this type can only be produced using a fixed set of constructions, each corresponding to a logical axiom or inference rule. The part of the system that provides these interfaces is called the *inference kernel*, or just the *kernel* of the theorem prover. It forms the *trusted computing base* of the system: If its soundness is guaranteed, then the system as a whole is also sound, as the kernel effectively checks all proofs. Using terminology from the operating systems world, this separates the code in a theorem prover into a trusted "kernel-space" and an untrusted "user-space".

The LCF approach is roughly equivalent to demanding that all provers must produce proofs in some explicit form, which are checked independently. Realizing the separation within the same system, means that the actual proofs never have to be stored explicitly (assuming proof irrelevance in the logic), although most systems also have an explicit representation for proofs.

The most obvious advantage of the LCF approach is safety. Keeping the soundness-critical code base small minimizes the possibility for errors in the implementation, and thus provides a higher level of assurance that the proofs produced by the system are correct. However, equally important is its extensibility: Since any extension in user-space is preserves soundness by construction, it may itself be entirely untrusted and still be used to construct proofs of a very high assurance. In particular, a change to such an extension does not affect the integrity of the kernel.

Like most of the proof automation in Isabelle, all proof procedures presented in this thesis are implemented in user-space. Hence they may be safely used even without trusting their author.

Definitional specification mechanisms While the LCF principle guarantees that only valid conclusions can be derived from the axioms, it cannot rule out the possibility that the axioms themselves are inconsistent, in which case anything could be derived from them.

It is therefore generally discouraged to extend the system with arbitrary axioms, and the preferred mode of usage is to leave the basic set of axioms untouched and extend the theory by definitions only. For example, we may define a new constant *One* :: *nat* by the following equation, where *Suc* denotes the successor of a natural number:

$$\mathit{One} = \mathit{Suc}\ \mathit{0}$$

Due to its special form *new-name* = *closed-term*, a definition cannot endanger consistency — after all, it is just an abbreviation and can always be eliminated by unfolding it. Hence, we can add definitions freely, while asserting arbitrary axioms requires a justification.

However, the definition scheme above is quite restrictive, and one might argue that the following are definitions as well:

(a) $\mathit{even}\ \mathit{0}$
$\mathit{even}\ n \Longrightarrow \mathit{even}\ (\mathit{Suc}\ (\mathit{Suc}\ n))$

(b) $$\begin{aligned} \mathit{even}\ \mathit{0} &= \mathit{True} \\ \mathit{even}\ (\mathit{Suc}\ n) &= \mathit{odd}\ n \\ \mathit{odd}\ \mathit{0} &= \mathit{False} \\ \mathit{odd}\ (\mathit{Suc}\ n) &= \mathit{even}\ n \end{aligned}$$

(c) $U\ x = U\ x + \mathit{1}$

The specification in (a) is an inductive definition of the predicate *even*. In (b) we have an alternative definition of the same predicate, but written in functional form, together with its companion *odd*. Finally, specification (c) is also recursive, but it is not an admissible definition: If $U\ x = U\ x + \mathit{1}$, then we may subtract $U\ x$ on both sides, and we can derive $\mathit{0} = \mathit{1}$ — an inconsistency. The reason is that this recursive equation has no total model, but all functions in HOL must be total.

Instead of extending the notion of definition to include definitions of the form (a) and (b) and carefully avoiding (c), the solution is to keep the restricted notion of definition, and reduce more complex ones to this primitive form. For example, the inductive definition of *even* can be reduced to the equation

$$\mathit{even} = \mathit{lfp}\ (\lambda p\ x.\ x = \mathit{0} \vee (\exists\, n.\ x = \mathit{Suc}\ (\mathit{Suc}\ n) \wedge p\ n)),$$

where *lfp* is a least fixed-point operator on the lattice of predicates over *nat*. The inductive specification given above can then be derived. Similarly, the mutual recursive definition of *even* and *odd* can be reduced to a primitive definition.

Reducing every single definition back to the basic principles using fixed-point theorems and recursion combinators is no less tedious than reducing all reasoning to primitive inference steps, and it is only practical because it can be fully automated. This automation is implemented in a *definitional specification mechanism*, also called a *definitional package* or just *package*. It automates the task of transforming a specification to a more primitive form, which is then introduced as a definition. Then, the original specification is derived from the primitive definition in a fully automated way.

Definitional packages take the LCF approach to the level of specifications. As all reasoning is formally checked and the notion of definition is not extended, definitional packages are conservative by construction, and thus offer a maximum of safety. At the same time they are convenient, as the internal constructions are transparent to the user.

In Isabelle/HOL, definitional packages are available for defining inductive predicates [89], inductive datatypes and primitive recursive functions [14], record types [80], and total recursive functions (the TFL package [97, 98]). The more recent nominal datatype package for defining data structures with binders [108, 109] is another example for this class of tools. In Chapter 2, we will develop a package for defining a more general class of functions than previously possible.

1.3 Isabelle/HOL

Isabelle [90, 81, 115] is a generic interactive theorem prover designed to be independent from concrete object logics. It provides a relatively weak meta-logic, called Isabelle/Pure, which serves as a natural deduction framework that can embed other logics.

Pure Isabelle/Pure is based on minimal higher-order logic. Types are built from type variables (α, β, γ, ...) and type constructor symbols of a fixed arity written in postfix notation like in ML. The special type constructor symbol $\Rightarrow$ is written infix and denotes the function space. Terms of type *prop* denote propositions, which can be formed using the built-in constants

$$\begin{array}{rll} \Longrightarrow :: \mathit{prop} \Rightarrow \mathit{prop} \Rightarrow \mathit{prop} & & \text{(implication)} \\ \bigwedge :: (\alpha \Rightarrow \mathit{prop}) \Rightarrow \mathit{prop} & & \text{(universal quantification)} \\ \equiv :: \alpha \Rightarrow \alpha \Rightarrow \mathit{prop} & & \text{(equality)} \end{array}$$

Specific object logics are embedded into the meta-logic by declaring their connectives as constants, and their characteristic axioms and inference rules as axioms in Pure. For example, the natural deduction rule for disjunction elimination

$$\frac{P \vee Q \qquad \begin{array}{c}[P]\\ \vdots \\ R\end{array} \qquad \begin{array}{c}[Q]\\ \vdots \\ R\end{array}}{R}$$

is represented in Isabelle/Pure as

$$P \vee Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$$

Thus, the rules of the object logic become theorems of the meta-logic.

In this thesis, we use the terms *rule* and *theorem* synonymously, and we sometimes write the outermost implication in inference rule notation when it improves readability.

Tactics and structured proofs Intuitively, *tactics* are proof procedures that operate on goals. They are composable from smaller tactics and range from single rule applications to full-blown automated reasoning procedures. Thus, goal-oriented proofs work backwards by first stating a goal and then repeatedly applying tactics until no more subgoals are left.

User-level proofs are written in the structured proof language Isar [114], a declarative formalism for managing contexts and the flow of facts. Isar only provides the abstract notation for structuring proofs, while most of the the actual reasoning is done by proof *methods*, which are invoked in the text. Simplifying a little, methods can be seen as tactics that are available at the Isar level. Commonly used methods are *simp* (simplification), *blast* (a classical prover based on tableaux), and *auto* (which tries to combine classical reasoning with rewriting and some arithmetic). The special-purpose proof procedures we develop in this thesis are made available to the user as methods.

When we give examples for user-level proofs, they are written in Isar. Since that language was designed to be human-readable and has a terminology that is close enough to common mathematical notation, we hope that readers unfamiliar with it are nevertheless able to read the proofs and follow their structure, even without further introduction.

We will frequently use the notion of *context*. For us, a context is a set of variables and assumptions that are present locally, and captures how structured statements are decomposed. For example, the proposition $\bigwedge a\ b\ c.\ P \Longrightarrow Q \Longrightarrow R$ corresponds to a context $\Gamma = \bigwedge a\ b\ c.\ P;\ Q$ and a conclusion R. Conversely, we write $\Gamma \Longrightarrow R$ for the above proposition. By abuse of notation, Γ may bind variables in R.

HOL: Basic logic Isabelle/HOL is the most widely used and best developed object-logic of Isabelle. It provides classical higher-order logic, based on Church's simple theory of types [29]. Apart from the support for type classes and overloading [113], Isabelle/HOL is compatible with the other major implementations of higher-order logic, namely HOL4 [44] and HOL Light [51]. Its semantics is given by a mapping into ZFC set theory [44].

The type *bool* is used for object-level propositions, with the object-level connectives $\wedge$, $\vee$, $\longrightarrow$, $\forall$, $\exists$, etc. Booleans are coerced to meta-level propositions using the embedding $\mathit{Trueprop} :: \mathit{bool} \Rightarrow \mathit{prop}$, which is usually hidden by the syntax layer. Object level equality is denoted by $= :: \alpha \Rightarrow \alpha \Rightarrow \mathit{bool}$.

HOL provides two choice operators: The definite description operator $\mathit{THE} :: (\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha$ works as a binder and is axiomatized as $(\mathit{THE}\ x.\ x = a) = a$. Hilbert's choice operator *SOME* is stronger, as it does not require the description to be unique. It is axiomatized by $P\ a \Longrightarrow P\ (\mathit{SOME}\ x.\ P\ x)$. We will not use the latter operator, but the former plays an important role when turning a relation into a function.

Since HOL is embedded into Pure and both languages have their own set of logical connectives ($\bigwedge$, $\Longrightarrow$, $\equiv$ vs. $\forall$, $\longrightarrow$, $=$), new users are often confused about when to use which. Fortunately, the intricacies arising from the differences between object- and meta-logic are irrelevant for this work, and the reader is invited to ignore the diffences between the two levels. All the methods that we develop work equally well in a system that has just one level of logic.

Inductive predicates and datatypes The inductive package [89] automates the definition of inductive predicates using the Knaster-Tarski fixed-point theorem. It provides the appropriate introduction, elimination, and induction rules. ML-style inductive datatypes are implemented by the datatype package [14], which also provides a combinator for primitive recursion over each newly-defined type. Many basic types like *list* are defined like this, and others like *nat* are characterized as datatypes a posteriori, which makes the datatype infrastructure available for them, too.

Products and Sums The types of products $\alpha \times \beta$ and sums $\alpha + \beta$ are so common that they deserve special mentioning. Values of product type are pairs, written in pair notation $(x,\ y)$. Arbitrary tuples are represented by nested products, which has the odd side effect that $(x,\ (y,\ z))$ and $(x,\ y,\ z)$ are indistinguishable.

Sum types denote disjoint unions, with the injections $\mathit{Inl} :: \alpha \Rightarrow \alpha + \beta$ and $\mathit{Inr} :: \beta \Rightarrow \alpha + \beta$. Case distinction over sums is expressed using the eliminator

$$\mathit{sum\text{-}case} :: (\alpha \Rightarrow \gamma) \Rightarrow (\beta \Rightarrow \gamma) \Rightarrow \alpha + \beta \Rightarrow \gamma$$

which satisfies the equations

$$\begin{aligned}&\mathit{sum\text{-}case}\ f\ g\ (\mathit{Inl}\ a) = f\ a\\&\mathit{sum\text{-}case}\ f\ g\ (\mathit{Inr}\ b) = g\ b\end{aligned}$$

Sets and Relations In the typed set theory of HOL, the type α *set* is just an abbreviation for the predicate type $\alpha \Rightarrow \mathit{bool}$, and hence $x \in S$ is equivalent to $S\ x$, and $\{x.\ P\ x\}$ is equivalent to P. Isabelle provides standard set-theoretic notation, which requires no further introduction. However, we should mention the syntax for general set comprehensions

$$\{f\ x\ y\ |x\ y.\ P\ x\ y\}$$

which corresponds to standard mathematical notation, except that the variables x and y are explicitly bound after the vertical bar. Internally, this notation expands to

$$\{u.\ \exists\, x\ y.\ u = f\ x\ y \wedge P\ x\ y\}\ .$$

Relations are represented as sets of pairs, and the composition of two relations is defined as follows:

$$R \circ S = \{(x,\ z).\ \exists\, y.\ (x,\ y) \in S \wedge (y,\ z) \in R\}$$

The argument order arises from the analogy to function composition. It looks counterintuitive at first, and the literature (especially in term rewriting) often uses the opposite argument order. However, we adopt the definition above, which is present in the Isabelle library.

Wellfoundedness A relation is wellfounded if it satisfies the principle of wellfounded induction:

$$\begin{aligned}&\mathit{wf}\ R =\\&(\forall\, P.\ (\forall\, x.\ (\forall\, y.\ (y,\ x) \in R \longrightarrow P\ y) \longrightarrow P\ x) \longrightarrow (\forall\, x.\ P\ x))\end{aligned}$$

An alternative characterization is that every nonempty set has a minimal element:

$$wf\ R = (\forall\ Q\ x.\ x \in Q \longrightarrow (\exists\, z{\in}Q.\ \forall\, y.\ (y,\ z) \in R \longrightarrow y \notin Q))$$

The most important property of a wellfounded relation is the wellfounded induction rule, which will be the basis of the function definitions in chapter 2:

$$\frac{wf\ r \qquad \bigwedge x.\ (\bigwedge y.\ (y,\ x) \in r \Longrightarrow P\ y) \Longrightarrow P\ x}{P\ a}$$

Since wellfounded relations are a critical ingredient for the function definitions and termination proofs, we briefly mention some basic construction principles for them. First of all, the empty relation is wellfounded, and wellfoundedness is preserved by subsets:

$$wf\ \emptyset$$
$$wf\ R \Longrightarrow S \subseteq R \Longrightarrow wf\ S$$

We can construct new wellfounded relations by taking the inverse image of an arbitrary mapping into some wellfounded relation:

$$\textit{inv-image}\ R\ f = \{(x,\ y).\ (f\ x,\ f\ y) \in R\}$$
$$wf\ R \Longrightarrow wf\ (\textit{inv-image}\ R\ f)$$

An important instance of this construction is given by the *measure* combinator, which takes the inverse image of the standard order on natural numbers:

$$\textit{measure} :: (\alpha \Rightarrow \textit{nat}) \Rightarrow (\alpha \times \alpha)\ \textit{set}$$
$$\textit{measure} = \textit{inv-image}\ \{(x,\ y).\ x < y\}$$

Wellfoundedness is not preserved by arbitrary unions, as we can easily see from the simple example $\{(\mathit{0},\ \mathit{1})\} \cup \{(\mathit{1},\ \mathit{0})\}$. However, under some conditions we may conclude that the union of two wellfounded relations is wellfounded. The following property will be very important for termination proofs:

Lemma 1.1 (Union Lemma). $wf\ R \Longrightarrow wf\ S \Longrightarrow S \circ R \subseteq R \Longrightarrow wf\ (R \cup S)$

Another important lemma also describes the interaction beween wellfoundedness, union and composition:

Lemma 1.2 (Composition Lemma). $wf\ (R \cup S) = wf\ (R \circ R \cup R \circ S \cup S)$

In particular, note the following consequence of this lemma (with $S \neq \emptyset$):

Lemma 1.3 (Self Composition Lemma). $wf\ R = wf\ (R \circ R)$

Reduction Pairs Our definition of wellfoundedness deals with strict relations $\prec$. Sometimes we must also talk about their weak counterparts $\preceq$, which are sometimes, but not always, the reflexive closure of $\prec$. Clearly, $\preceq$ cannot be wellfounded in our sense, as it is usually reflexive. The relevant property is that $\preceq$ is compatible with $\prec$ in the sense that $\preceq \circ \prec \subseteq \prec$ holds. This is formalized in the notion of a reduction pair:

$$\textit{reduction-pair}\ (R,\ S) = (wf\ R \wedge S \circ R \subseteq R)$$

Inverse images preserve this property, so we can build new reduction pairs using inverse images and measures:

$$\begin{aligned}&\textit{rp-inv-image}\ (R,\ S)\ f = (\textit{inv-image}\ R\ f,\ \textit{inv-image}\ S\ f)\\&\textit{reduction-pair}\ (R,\ S) \Longrightarrow \textit{reduction-pair}\ (\textit{rp-inv-image}\ (R,\ S)\ f)\\&\textit{measure-rp} = \textit{rp-inv-image}\ (\{(x,\ y).\ x < y\},\ \{(x,\ y).\ x \le y\})\end{aligned}$$

Lists and Multisets Lists are defined as an inductive datatype with the constructors *Nil* ([]) and *Cons* (:). The function $\textit{set} :: \alpha\ \textit{list} \Rightarrow \alpha\ \textit{set}$ forms the set of the elements of a list. The type α *multiset* formalizes finite multisets, which we write as $\{\!\!\{\ x,\ y,\ z\ \}\!\!\}$. Multiset union is denoted by $+$. The function $\textit{set-of} :: \alpha\ \textit{multiset} \Rightarrow \alpha\ \textit{set}$ converts multisets into sets.

1.4 Functional Programming in HOL

The similarities of higher-order logic and functional programming become evident when we look at the definition of a recursive function, such as list reversal:

$$\textit{rev} :: \alpha\ \textit{list} \Rightarrow \alpha\ \textit{list}$$

$$\begin{aligned}&\textit{rev}\ [\,] &&= [\,]\\&\textit{rev}\ (x{:}\textit{xs}) &&= \textit{rev}\ \textit{xs}\ @\ [x]\end{aligned}$$

This HOL definition is identical to the corresponding Haskell program, except for a minor syntactic variation: List concatenation is written @ instead of ++.

Likewise, datatype definitions are very similar in both worlds, and the type system of HOL, ML, and Haskell are based on the same foundations. Isabelle also has a notion of type classes [113] that has recently been extended to match Haskell type classes more closely [49].

Exploiting these similarities, Isabelle provides a code generator that translates HOL specifications to functional programs in one out of several supported target languages (currently SML, Haskell and OCaml are supported). The code generator was developed by Haftmann [48], extending earlier work by Berghofer [13].

This suggests a simple approach to verifying functional programs by representing them directly as recursively defined HOL functions. The properties of interest can then be proved in Isabelle, and code generation turns the specification into an executable program. As sloganized by Nipkow [81]:

$$\textit{HOL} = \textit{Functional Programming} + \textit{Logic.}$$

However, despite the striking similarities on the surface, functional programs and HOL specifications are fundamentally different in some important aspects.

Operational semantics: Evaluation The operational semantics of a functional language is typically expressed as an evaluation relation $\rightarrow_P$, which defines how terms are evaluated in the context of a program P. This relation specifies the order of evaluation (strict or lazy) and defines the operational meaning of the language primitives. There is no such operational semantics for higher-order logic, other than the β-reduction of the underlying lambda-calculus, which is far too restricted to be called a programming language.

What comes closest to evaluation in HOL is equational rewriting, also known as *simplification*. Given a set of equations, Isabelle's simplifier rewrites a term s repeatedly, until a normal form t is reached, which produces a proof of $s = t$. Unlike in functional languages, this form of simplification can rewrite with equations of arbitrary origin and form, as opposed to just using the defining equations, which have a restricted form. For example, the associativity law $(a + b) + c = a + (b + c)$ cannot be used as a defining equation in a functional program, but poses no problems to the simplifier.

Logically, the code generation facilities described above are nothing more than a variation of simplification that is less general but faster, as it uses the programming language implementation as a reduction engine. This evaluation mechanism is sound in the sense that any evaluation in the target language corresponds to a chain of rewriting steps in HOL [48].

Note that the generated code is evaluated with the evaluation strategy of the target language. This means that the same HOL term may exhibit very different behaviour depending on the target language. For example, if *ack* is the Ackermann function and *take n xs* extracts the first n elements from a list xs, the term *take 2* (*map* ($\lambda n.$ *ack n n*) [*1*, *2*, *15*]) will quickly yield a result when evaluated in Haskell, while ML will get lost in the evaluation of *ack 15 15*, which is not needed for the result. We can also construct examples that terminate under lazy evaluation and run forever when evaluated strictly. Since evaluating with different strategies may lead to different termination behaviour, there is no fixed notion of termination in higher-order logic. This will be particularly important in Chapter 3.

Denotational Semantics: Function spaces In the standard model, the HOL function type $\sigma \Rightarrow \tau$ includes all total functions from σ to τ in the set-theoretic sense. Excluding partial functions ensures that $f\ x$ is meaningful for any x. As there is no built-in notion of undefinedness, partiality must be modelled explicitly. This can be done using relations instead of functions, option types or underspecification, which we discuss below. A survey of the modelling techniques for partiality in logics of total functions is given by Müller and Slind [78].

Denotational semantics for functional languages is commonly described using notions from domain theory [2]. Here, terms denote values of *domains*, which are complete partial orders (cpos), where the ordering relation $\sqsubseteq$ intuitively means "less defined or equal to". The least element $\perp$ denotes a diverging computation.

In domain theory the function space $\sigma \rightarrow \tau$ denotes the type of continuous functions between the domains of σ and τ. A model of a recursive definition is then given as a least solution of a fixed-point equation.

For example, the recursive SML definition

```
fun f (n : int) = (if n = 3 then 1 else f (n + 1))
```

denotes the least fixedpoint of the functional F, defined as $F\ f\ x = (\textit{if}\ n = 3\ \textit{then}\ 1\ \textit{else}\ f\ (n + 1))$, which results in the function

$$f(n) = \begin{cases} 3 & \text{if } n \leq 3 \\ \perp & \text{otherwise.} \end{cases}$$

There is no equivalent HOL function, since there exists no value $\bot :: int$ other than the integers, unless a flat domain is modelled explicitly, as it is done in HOLCF [77]. However, this particular fixed-point equation has other solutions, as we shall see below.

Modelling Partiality by Underspecification Underspecification means to replace a partial function by an arbitrary total completion. We can model the partial function f above by introducing an unspecified constant $u :: int$. Such an unspecified value is always safe to introduce, and we can use it to define f by $f\ n = (\textsf{if}\ n \leq 3\ \textsf{then}\ 3\ \textsf{else}\ u)$. Now we can prove that this function satisfies the recursive equation above, but it is still underspecified, since we cannot determine the value of $f\ 4$ from the specification. In fact, f stands for a whole class of functions.

Underspecification can be a convenient device for handling partiality, but the unspecified value u that we introduced does not always behave like $\bot$: For example, in order to show that $f\ 3$ is defined, we might like to show that $f\ 3 \neq u$. However, since u is unspecified, it may well be equal to 3, and thus this statement is unprovable.

The discussion above may lead to the impression that HOL is strictly less expressive than functional languages, due to the lack of partial functions. However we can also express things in HOL that cannot be written in functional languages, since they are not programs. One of the shortest examples is probably the following function, where $xs_{[i]}$ denotes the i-th element of list xs, and $concat$ denotes concatenation of a list of lists:

$$\begin{aligned}&pcp\ xs\ ys =\\&(\exists is.\ is \neq [\,] \wedge\\&\qquad concat\ (map\ (\lambda i.\ xs_{[i]})\ is) = concat\ (map\ (\lambda i.\ ys_{[i]})\ is))\end{aligned}$$

The function describes Post's Correspondance Problem, which is undecidable.

1.5 Structure of this thesis

Although all research presented in this thesis is driven by just one goal — improving the tool support for function definitions — the three main chapters are essentially independent. They cover three main aspects of the problem: constructing functions (ch. 2), proving their termination (ch. 3), and handling pattern matching (ch. 4). The tools and ideas developed there are then used to build another tool (ch. 5), which is not directly related to function definitions, but targets the automatic derivation of induction schemes.

To economize on the reader's cognitive stack space, each of the main chapters comes with its own introduction and discussion of related work.

One last warning This presentation does not always show our tools in exactly the same way as they were implemented. Rather, it describes a certain idealized state, to which the implementation should eventually converge. Reality is always different, and readers who intend to work with the implementation on a level below the standard user interface should be aware of this. Nevertheless, all presented methods are real, and the differences to the implementation are only minor.

Chapter 2

Function definitions

Contents

2.1 Introduction

In this chapter, we describe the foundations and mechanics of a definition principle for partial recursive functions in Isabelle/HOL. From a recursive specification, our package defines a partial function (modeled as an underspecified total function), together with a set describing the function's domain. On the domain, the defined function coincides with the specification. The provided proof rules allow convenient reasoning about such partial functions, as is common practice for total functions [81].

As a pleasant side effect of handling partiality, our approach naturally supports nested recursive definitions, which have posed technical problems for a long time [99, 64]. Most of these difficulties disappear entirely in our setting, since the explicit domain cleanly separates partial correctness from termination properties.

2.1.1 Motivation

Partiality

As an example of a partial recursive function, we define an interpreter for a minimalistic imperative language. Such an interpreter must be partial, since the interpreted program might loop and this non-termination cannot be detected.

However we would expect to be able to prove termination for certain classes of programs — for example, the class of all programs without while loops.

The language is modelled in a straightforward manner. Variable names and values are simply natural numbers, and environments map variables to values. For simplicity, a shallow embedding is used for expressions, instead of modeling their syntax. They are just mappings from environments to values:

types
  $var = nat$
  $val = nat$
  $env = var \Rightarrow val$
  $exp = env \Rightarrow val$

The datatype of commands is defined as follows:

datatype $com =$
  $Assign\ var\ exp$
$|\ Seq\ com\ com$
$|\ If\ exp\ com\ com$
$|\ For\ exp\ com$
$|\ While\ exp\ com$

The execution of a command yields a transformation on the environment. In the equations below, $f(x := y)$ denotes function update, and *fun-pow* denotes function exponentiation.

$exec :: com \Rightarrow env \Rightarrow env$

$exec\ (Assign\ v\ exp)\ e = e(v := exp\ e)$
$exec\ (Seq\ c_1\ c_2)\ e = exec\ c_2\ (exec\ c_1\ e)$
$exec\ (If\ exp\ c_1\ c_2)\ e = \textsf{if}\ exp\ e \neq 0\ \textsf{then}\ exec\ c_1\ e\ \textsf{else}\ exec\ c_2\ e$
$exec\ (For\ exp\ c)\ e = fun\text{-}pow\ (exp\ e)\ (exec\ c)\ e$
$exec\ (While\ exp\ c)\ e = \textsf{if}\ exp\ e = 0\ \textsf{then}\ exec\ (While\ exp\ c)\ (exec\ c\ e)\ \textsf{else}\ e$

In Isabelle 2005 and earlier, the definition of *exec* cannot be made. The attempt leads to an unsolvable termination proof obligation.

As a workaround, we can always extend a partial function to a total one: If we know that the function terminates under certain conditions, this check can be added to the function body, returning a dummy value if the check fails:

$$f\ x = (\textsf{if}\ \langle guard\rangle\ \textsf{then}\ \langle body\rangle\ \textsf{else}\ dummy)$$

Then f can be defined as a total function. But this is unsatisfactory as a general method for two reasons:

First, the termination guard must be known at definition time. If it turns out later that this condition was too restrictive, we must change the definition of the function. Restricting the definition of our interpreter to a certain subclass of terminating programs (e.g., programs without while loops) is certainly inadequate. So we would have to find a condition that covers all possible terminating programs, which is at least not obvious, and certainly not always executable.

Second, the workaround changes the body of the function. Introducing the termination guard, which is alien to the functional specification, is inelegant and may cause difficulties when executable code is to be extracted from the definition at a later stage.

In contrast, our package allows to define *exec* as a partial function and later prove its termination on the values we need.

Nested recursion

Functions with nested recursive calls are notoriously difficult to define and reason about. The central problem is that the termination proof for such functions requires some reasoning about partial correctness properties beforehand.

As a classic example, consider the following definition of the constant zero function on natural numbers:

$$Z\ n = (\textsf{\textit{if}}\ n = \mathit{0}\ \textsf{\textit{then}}\ \mathit{0}\ \textsf{\textit{else}}\ Z\ (Z\ (n - \mathit{1})))$$

For the termination of the outer call, we would usually prove that $n \neq \mathit{0} \Longrightarrow Z\ (n - \mathit{1}) < n$. Since the function always returns zero, this is certainly true, but seems difficult to prove before the function is "properly" defined. We can identify two problems here:

1. If the system requires the termination proof to be conducted before the function symbol Z is even introduced in the logic, it is difficult to support nested recursion, since the termination goal can not even be stated. Definitional packages do not have this problem, since definitions are transformed into a non-recursive form and can be introduced into the logic immediately without proof. The challenge is then to derive the desired properties afterwards.

2. After stating the termination goal, we need to prove it, and this requires reasoning principles for the function. But the main tool, namely functional induction, is usually not available at that point, since it depends on the termination of the function. We solve this by making a restricted version of functional induction available from the very beginning. Using that rule, proofs for partial correctness properties are simple and natural.

2.1.2 An overview of the approach

Starting from the specification of a function *f*, the package inductively defines its graph G_f and its domain dom_f, following the recursive structure of the definition. Using the definite description operator, the graph is turned into a total function *f*, which models the specified partial function on the domain.

Then the package proves that G_f actually describes a function on dom_f, i.e. that function values exist and are unique. Then it automatically derives the original recursion equations and an induction rule. The rules are constrained by premises of the form $t \in dom_f$, that is, they describe the function's behaviour on its domain only. Despite these constraints, they allow convenient reasoning about the function, even before its termination is established. To support natural termination proofs, the package provides a special termination rule, in addition to the domain rules.

2.2 The Process of Definition

This section describes the definitional core of the package. We ignore extra features like pattern matching, currying and mutual recursion for now and restrict our attention to the essential ingredients. Extensions will be discussed in §2.6.

We start with the recursive specification given as input by the user as a fixed point equation of a functional F, which he would like to get back as a theorem in the end:

$$f\,x = F\,f\,x$$

2.2.1 Recursive calls in a higher-order setting

First of all, we need to analyze the definition and extract the recursive calls.

A recursive call can be written $[\Gamma \leadsto r]$, where r is the argument of the call, and Γ is a context that specifies when the call occurs. In general, Γ can contain both bound variables and assumptions.

For example, the definition

$$f\,n = (\textsf{if}\ n = 0\ \textsf{then}\ 0\ \textsf{else}\ f\ (n - 1))$$

has a recursive call $[n \neq 0 \leadsto n - 1]$.

If we have higher-order recursion, the case becomes more complicated. Consider a datatype of n-ary trees:

$$\textbf{datatype}\ \alpha\ \mathit{tree} = \mathit{Node}\ \alpha\ (\alpha\ \mathit{tree}\ \mathit{list})$$

We can define a function $\mathit{mirror} :: \alpha\ \mathit{tree} \Rightarrow \alpha\ \mathit{tree}$ as follows:

$$\mathit{mirror}\ (\mathit{Node}\ a\ \mathit{ts}) = \mathit{Node}\ a\ (\mathit{map}\ \mathit{mirror}\ (\mathit{rev}\ \mathit{ts}))$$

Here, *mirror* is passed as an argument to *map*, so we do not see immediately what the argument of the recursive call is. The answer is that the call can be described by $[\bigwedge x.\ x \in \mathit{set}\ (\mathit{rev}\ \mathit{ts}) \leadsto x]$. This means that calls may occur at any element of the list *rev ts*. Of course, some knowledge about *map* is required to come up with this description of the recursive call.

For the extraction of recursive calls, we use a procedure due to Slind [97, 98], which employs congruence rules to deal with higher-order recursion. The details of this extraction process are not relevant for the understanding the rest of the package, and we defer the description of the algorithm to §2.5. For the moment, we care only about the property that the recursive calls must satisfy:

Definition 2.1 (Congruence Condition)**.** *For a functional F and the recursive calls $[\Gamma_1 \leadsto r_1], \ldots, [\Gamma_k \leadsto r_k]$, the* congruence condition *is the implication*

$$\begin{aligned}&(\Gamma_1 \Longrightarrow f\,r_1 = f'\,r_1) \Longrightarrow\\ &\ldots \Longrightarrow (\Gamma_k \Longrightarrow f\,r_k = f'\,r_k) \Longrightarrow F\,f\,x = F\,f'\,x\,.\end{aligned}$$

Intuitively, this condition states that it is enough to know how a function f behaves at the recursive calls in order to compute $F\,f\,x$. The values of f on all other inputs do not influence the result. So we assume for now that we have a procedure to extract calls from the recursive equation, such that the congruence condition holds.

The congruence condition is always trivially satisfied by the single recursive call $[\bigwedge x. \leadsto x]$, where it degenerates into the vacuous

$$(\bigwedge x.\ f\,x = f'\,x) \Longrightarrow F\,f\,x = F\,f'\,x.$$

We could actually define any function from this, but it would be of little use: As we will see, the set of recursive calls also determines the definition of the *domain* of the function. In this case, the domain would always be empty.

2.2.2 Defining the graph, the function, and the domain

From the functional specification, we generate the inductive definition of a relation G_f, which represents the graph of the function. Assume that we have extracted the recursive calls $[\Gamma_1 \leadsto r_1], \ldots, [\Gamma_k \leadsto r_k]$. The relation G_f is then defined inductively by the following rule:

$$\frac{(\Gamma_1^{[h/f]} \Longrightarrow (r_1^{[h/f]}, h(r_1^{[h/f]})) \in G_f)\ \ldots\ (\Gamma_k^{[h/f]} \Longrightarrow (r_k^{[h/f]}, h(r_k^{[h/f]})) \in G_f)}{(x,\ F\ h\ x) \in G_f}\,(GI)$$

In $\Gamma^{[h/f]}$ etc., the function variable h is substituted for the function symbol f. Intuitively, we add the pair $(x,\ F\ h\ x)$ to G_f for a fresh function variable h. The premises express that h coincides with G_f on all recursive calls.

Compared with a naive relational description, which would invent a new variable for the result of each recursive call, we use a single function variable h, which is constrained to the graph on all recursive calls. Inventing separate variables for the recursive calls would require additional bookkeeping and lead to problems with higher-order recursion.

After introducing G_f using the package for inductive definitions, we can already define the function f itself, using HOL's definite description operator:

$$f = (\lambda x.\ \mathit{THE}\ y.\ (x,\ y) \in G_f)$$

We now have the definition, but it is not yet usable. We need to prove that it actually satisfies the specification. An important reasoning tool will be the *domain* of the function. It is defined inductively, too:

$$\frac{\Gamma_1 \Longrightarrow r_1 \in \mathit{dom}_f \qquad \ldots \qquad \Gamma_k \Longrightarrow r_k \in \mathit{dom}_f}{x \in \mathit{dom}_f}\ (\mathit{dom}_f\text{-}\mathit{intro})$$

This definition is structurally similar to the definition of G_f, but it is simpler, since it only talks about the function arguments, not the values. Also note that the function variable h is not used here, since f is already defined at this point. (Recall that in the case of nested recursion, some of the recursive calls may mention f.)

2.2.3 The relation is a function

We now have to show that the relation G_f describes a function on dom_f:

$$x \in \mathit{dom}_f \Longrightarrow \exists!y.\ (x,\ y) \in G_f$$

The proof of this property is performed automatically for each definition. The following proof sketch illustrates the structure of the derivation:

We use induction on dom_f. For some fixed $x \in dom$, the induction hypothesis ensures that the property holds on all recursive calls. Splitting into existence and uniqueness, and using the fact that the unique value is denoted by *f*, we get for each recursive call $[\Gamma_i \leadsto r_i]$:

$$\begin{array}{ll} \Gamma_i \Longrightarrow (r_i,\, f\, r_i) \in G_f & (\mathit{ihyp\text{-}ex}_i) \\ \bigwedge z.\ \Gamma_i \Longrightarrow (r_i,\, z) \in G_f \Longrightarrow z = f\, r_i & (\mathit{ihyp\text{-}un}_i) \end{array}$$

Now, the $\mathit{ihyp\text{-}ex}_i$ are exactly the premises the introduction rule for G. Hence we get the existence part $(x,\ F\ f\ x) \in G$.

For the uniqueness part, we must show that this is the only possible value of the function. We assume another y with $(x,\ y) \in G$. By inversion on G we know that $y = F\ h\ x$ for some h, and that h follows G on the recursive calls. Hence we get $\Gamma_i \Longrightarrow h\ r_i = f\ r_i$ for each recursive call. It remains to apply the congruence property and conclude that $F\ f\ x = F\ h\ x$, which proves uniqueness.

2.2.4 Deriving simplification and induction rules

Having established that function values exist and are unique on the domain, we prove the original recursion equation and an induction rule. The equation is just as given in the original specification, but guarded by a domain condition:

$$x \in dom_f \Longrightarrow f\ x = F\ f\ x$$

Deriving the recursion equation is now simple: From uniqueness we know that $(x,\ y) \in G_f$ implies $f\ x = y$, and we have already proved the required relation in the existence part of the previous proof. We can reuse it after lifting it out of the induction context, which is technical but straightforward.

The partial induction rule follows the structure of the recursion: In each case, the property may be assumed on the arguments of the recursive calls, but the final inductive result is restricted to dom_f:

$$\frac{\begin{array}{c}\bigwedge x.\ x \in dom_f \Longrightarrow (\Gamma_1 \Longrightarrow P\ r_1) \Longrightarrow \ldots \\ \Longrightarrow (\Gamma_k \Longrightarrow P\ r_k) \Longrightarrow P\ x\end{array}}{a \in dom_f \Longrightarrow P\ a}\ (\mathit{pinduct}_f)$$

This rule is very similar to the induction principle that comes with dom_f, which looks like this:

$$\frac{\begin{array}{c}\bigwedge x.\ (\Gamma_1 \Longrightarrow r_1 \in dom_f) \Longrightarrow (\Gamma_1 \Longrightarrow P\ r_1) \Longrightarrow \ldots \\ \Longrightarrow (\Gamma_k \Longrightarrow r_k \in dom_f) \Longrightarrow (\Gamma_k \Longrightarrow P\ r_k) \Longrightarrow P\ x\end{array}}{a \in dom_f \Longrightarrow P\ a}\ (dom_f\text{-}\mathit{induct})$$

The first rule is easily derived from the second, but it is better suited for automation, since the premise $x \in dom_f$ is useful to unfold a call $f\ x$, which would normally occur in an actual induction.

The induction rule typically does not mention the function *f*. In a concrete induction, the function usually occurs in the instantiation of P. However, when the recursion is nested, then the function already appears in the induction rule, in some of the Γ_i or r_i.

With the proof of the partial simplification and induction rules, the actual definition process is completed: The rules provide adequate means for reasoning about the function. In particular, we can now establish the properties we might need for a termination proof. We will see in §2.4 that this is very useful when dealing with nested recursion.

2.2.5 Simple Examples

The following simple examples illustrate the behaviour of the package. For each function, we give the extraced recursive calls, the definitions of the graph and the domain, and the generated simplification and induction rules.

The Fibonacci function

Recursive equation:

$$\mathit{fib}\ n = (\textsf{if}\ n \leq 1\ \textsf{then}\ n\ \textsf{else}\ \mathit{fib}\ (n - 1) + \mathit{fib}\ (n - 2))$$

Extracted calls:

$$[\neg\ n \leq 1 \leadsto n - 1], [\neg\ n \leq 1 \leadsto n - 2]$$

Graph:

$$\frac{\begin{array}{c}\neg\ n \leq 1 \Longrightarrow (n - 1,\ h\ (n - 1)) \in G_{\mathit{fib}} \\ \neg\ n \leq 1 \Longrightarrow (n - 2,\ h\ (n - 2)) \in G_{\mathit{fib}}\end{array}}{(n,\ \textsf{if}\ n \leq 1\ \textsf{then}\ n\ \textsf{else}\ h\ (n - 1) + h\ (n - 2)) \in G_{\mathit{fib}}}$$

Domain:

$$\frac{\neg\ n \leq 1 \Longrightarrow n - 1 \in \mathit{dom}_{\mathit{fib}} \qquad \neg\ n \leq 1 \Longrightarrow n - 2 \in \mathit{dom}_{\mathit{fib}}}{n \in \mathit{dom}_{\mathit{fib}}}$$

Simplification and induction rules:

$$\begin{array}{l}\ \ n \in \mathit{dom}_{\mathit{fib}} \Longrightarrow \\ \mathit{fib}\ n = (\textsf{if}\ n \leq 1\ \textsf{then}\ n\ \textsf{else}\ \mathit{fib}\ (n - 1) + \mathit{fib}\ (n - 2))\end{array}$$

$$\frac{\begin{array}{l}\bigwedge n.\ n \in \mathit{dom}_{\mathit{fib}} \Longrightarrow \\ \qquad (\neg\ n \leq 1 \Longrightarrow P\ (n - 1)) \Longrightarrow \\ \qquad (\neg\ n \leq 1 \Longrightarrow P\ (n - 2)) \Longrightarrow P\ n\end{array}}{a \in \mathit{dom}_{\mathit{fib}} \Longrightarrow P\ a}$$

Nested zero

We now define the nested zero function from §2.1.1. Observe that the nested recursion does not make our definitions circular. The definition of dom_Z below may refer to Z, which is already defined.

Recursive equation:

$$Z\ n = (\textsf{if}\ n = 0\ \textsf{then}\ 0\ \textsf{else}\ Z\ (Z\ (n - 1)))$$

Extracted calls:

$$[n \neq 0 \leadsto n - 1], [n \neq 0 \leadsto Z\ (n - 1)]$$

Graph:

$$\frac{\begin{array}{c} n \neq 0 \Longrightarrow (n - 1,\ h\ (n - 1)) \in G_Z \\ n \neq 0 \Longrightarrow (h\ (n - 1),\ h\ (h\ (n - 1))) \in G_Z \end{array}}{(n,\ \textsf{if}\ n = 0\ \textsf{then}\ 0\ \textsf{else}\ h\ (h\ (n - 1))) \in G_Z}$$

Domain:

$$\frac{n \neq 0 \Longrightarrow n - 1 \in dom_Z \qquad n \neq 0 \Longrightarrow Z\ (n - 1) \in dom_Z}{n \in dom_Z}$$

Simplification and induction rules:

$$n \in dom_Z \Longrightarrow Z\ n = (\textsf{if}\ n = 0\ \textsf{then}\ 0\ \textsf{else}\ Z\ (Z\ (n - 1)))$$

$$\frac{\begin{array}{l} \bigwedge n.\ n \in dom_Z \Longrightarrow \\ \qquad (n \neq 0 \Longrightarrow P\ (n - 1)) \Longrightarrow \\ \qquad (n \neq 0 \Longrightarrow P\ (Z\ (n - 1))) \Longrightarrow P\ n \end{array}}{a \in dom_Z \Longrightarrow P\ a}$$

The partial function findzero

The function *findzero* $(f,\ n)$ returns the smallest value $n' \geq n$ such that $f\ n' = 0$. If no such value exists, the function diverges.

Recursive equation:

$$findzero\ (f,\ n) = (\textsf{if}\ f\ n = 0\ \textsf{then}\ n\ \textsf{else}\ findzero\ (f,\ Suc\ n))$$

Extracted calls:

$$[f\ n \neq 0 \leadsto (f,\ Suc\ n)]$$

Graph:

$$\frac{f\ n \neq 0 \Longrightarrow ((f,\ Suc\ n),\ h\ (f,\ Suc\ n)) \in G_{fz}}{((f,\ n),\ \textsf{if}\ f\ n = 0\ \textsf{then}\ n\ \textsf{else}\ h\ (f,\ Suc\ n)) \in G_{fz}}$$

Domain:

$$\frac{f\ n \neq 0 \Longrightarrow (f,\ Suc\ n) \in dom_{fz}}{(f,\ n) \in dom_{fz}}$$

Simplification and induction rules:

$$\begin{array}{l} (f,\ n) \in dom_{fz} \Longrightarrow \\ findzero\ (f,\ n) = (\textsf{if}\ f\ n = 0\ \textsf{then}\ n\ \textsf{else}\ findzero\ (f,\ Suc\ n)) \end{array}$$

$$\frac{\bigwedge f\ n.\ (f,\ n) \in dom_{fz} \Longrightarrow (f\ n \neq 0 \Longrightarrow P\ (f,\ Suc\ n)) \Longrightarrow P\ (f,\ n)}{(f',\ n') \in dom_{fz} \Longrightarrow P\ (f',\ n')}$$

The function that is always undefined

The package even lets us define the function that never terminates. However, the results are not very interesting:

Recursive equation:

$$U\ x = U\ x + 1$$

Extracted calls:

$$[\ \rightsquigarrow x]$$

Graph:

$$\frac{(x,\ h\ x) \in G_U}{(x,\ h\ x + 1) \in G_U}$$

Domain:

$$\frac{n \in dom_U}{n \in dom_U}$$

Simplification and induction rules:

$$x \in dom_U \Longrightarrow U\ x = U\ x + 1$$

$$\frac{\bigwedge x.\ x \in dom_U \Longrightarrow P\ x \Longrightarrow P\ x}{a \in dom_U \Longrightarrow P\ a}$$

Both the graph and the domain are just the empty set, and hence the simplification and induction rules are just instances of *ex falso quodlibet.* The inconsistency of the equation $U\ x = U\ x + 1$ is avoided by the precondition $x \in dom_U$.

2.3 Termination Proofs

All results obtained from the partial simplification and induction rules will contain domain conditions of the form $t \in dom_f$. It is thus desirable to prove more about dom_f, which is the objective of a termination proof. Often, our goal will be to show that a function is total, that is, any value is element of dom_f. For partial functions, we will usually be interested in a certain subset.

While the definition process we saw above is fully automated and works for any function definition, we cannot expect such complete automation for termination proofs, since termination is undecidable.

Our primary goal here is to provide a clear and simple interface for integrating automated tools without requiring a deep knowledge of the inner workings of the function package. Moreover, since automated tools can always fail, we also need to provide a simple interface to the user, who may need to perform the termination proof interactively. TFL did not provide such an interface, which required artificial workarounds when the termination proof required user intervention.

2.3.1 Elementary proofs, using the definition of the domain

The definition of the domain that we gave in the previous section is already a very natural description of the termination behaviour of the function. It is not hard to do a termination proof with just the domain introduction rule and a suitable induction principle.

For example, we can show that *fib* is total by simple complete induction, using the domain introduction rules. Here is a detailed manual proof in Isar:

lemma *fib-terminates*: $n \in$ *fib-dom*
proof (*induct n rule*: *less-induct*) — Induction over the natural numbers
  fix *n* **assume** *IH*: $\bigwedge m.\ m < n \Longrightarrow m \in$ *fib-dom*

  show $n \in$ *fib-dom*
    by (*rule fib-dom.intros*) (*auto intro*: *IH*)
qed

2.3.2 Termination proofs using relations

The standard technique for proving that a function is total, is to provide a wellfounded relation R and show that all recursive calls are decreasing w.r.t. R. In previous approaches, the specified relation was also used for the actual function definition. Although our definition principle does not require user-specified relations, we can still support them as a way to prove termination. This is acomplished by the following rule, provided by our package for *fib*:

$$\frac{\begin{array}{c} \mathit{wf}\ R \\ \bigwedge n.\ \neg\ n \le 1 \Longrightarrow (n - 1,\ n) \in R \qquad \bigwedge n.\ \neg\ n \le 1 \Longrightarrow (n - 2,\ n) \in R \end{array}}{\forall\, x.\ x \in \mathit{dom}_{\mathit{fib}}}$$

With this rule, a termination proof can be done by giving a wellfounded relation and a proof that every recursive call is decreasing. Of course, this is just what is going on in the induction proof above, but here the induction is implicit in the relation R. This makes it easier to automate the proof, as the decrease conditions are often inequalities that are relatively easy to show once a suitable relation is found.

Here is the general form of this rule, which we call the *termination rule*:

$$\frac{\mathit{wf}\ R \qquad \bigwedge x.\ \Gamma_1 \Longrightarrow (r_1,\ x) \in R \qquad \ldots \qquad \bigwedge x.\ \Gamma_k \Longrightarrow (r_k,\ x) \in R}{\forall\, x.\ x \in \mathit{dom}_f}$$

The proof of the termination rule is just a wellfounded induction over R, performed automatically for each function definition.

The user usually applies the termination rule indirectly via a custom method called *relation*, which instantiates the rule with a user-specified relation. Proving the resulting inequalities is a one-liner if the function is as simple as *fib*. To establish its termination using the relation *less-than* $= \{(x,\ y).\ x < y\}$, the user just writes

termination *fib* — set up termination goal
by (*relation less-than*) *auto* — prove it

2.3.3 Simplification and induction rules revisited

When we have proved that the function is total, the domain conditions in the recursive equation and induction rule become obsolete, once the termination proof is finished. It is now easy to project them away, and we obtain what we call the *total* recursive equations and the *total* induction rule. These rules become the primary tools for reasoning about the function.

For partial functions, we can replace the abstract domain dom_f by a concrete set D, for which we have proved termination. Note that in order to replace dom_f in the *premises* of the induction rule, we must also show that D is downward closed under R, since the induction principle is only valid if calls on elements of D only recurse on elements of D.[1] In practice, this is often simple. For partial functions it is however often easier to derive the right induction rule directly using the method that will be describen in Chapter 5.

2.3.4 Integration of automated tools

Since proving termination of a function is just like proving a lemma, we have a clear interface for integrating automated termination provers. What we require is just a tactic that is able to solve goals of a certain form, given by the premises of the termination rule. In chapter 3, we will show that it is indeed possible to implement nontrivial termination provers as tactics.

2.4 Nested Recursion

Nested recursive definitions have recursive calls that depend on the results of other recursive calls. Thus, we usually need to prove some property about the behviour of the function before we can establish its termination. Here we are facing an apparent circularity, since the induction rule that we would like to use depends on termination.

Slind's TFL provides a "provisional induction rule" [99] to solve nested termination goals. This rule is basically a severely mangled functional induction rule, where the unsolved termination conditions become part of the function body. With TFL's second definition principle, relationless definition, this becomes even more difficult. The provisional induction rule can help with termination proofs, but this is often quite inelegant due to the structure of the rule. Slind already observes these shortcomings [99]:

> We regard our results on relationless definition of nested recursion as only partly satisfactory. The specified recursion equations and induction theorems are automatically derived, which is good; however, the termination proof using the provisional induction theorem and recursion equations for the auxiliary function is usually clumsy and hard to explain.

[1]For example, we cannot use the set of even numbers in our Fibonacci example. Although the function does terminate on all even numbers, the modified induction principle would be invalid.

As an alternative approach, Krstić and Matthews [64] propose the notion of *inductive invariants* to describe properties of a function f in terms of an input-output relation, without the need to explicitly mention f. They show how such an inductive invariant can be used to prove f's termination.

But this comes at a high cost, since establishing an inductive invariant is comparatively hard: The proof of an inductive invariant corresponds to a well-founded induction, and to be able to apply the induction hypothesis, we must show that the arguments in the inner recursive calls are decreasing. This means that we must anticipate parts of the termination proof to establish the inductive invariant.

Instead, we would like to be able to use functional induction, which is generally simpler[2]. Giesl [41] shows that this approach is sound: We may prove lemmas by functional induction and then use them (in a certain way) in the termination proof of the same function. The argument is that anything proved by functional induction is "partially true", i.e., it holds for all values for which the function terminates. Then, a close look reveals that at the positions where the lemmas are needed, we know that this condition holds, since the inner recursive calls are proved first. But since Giesl's informal proofs include statements like "P holds for all x where f terminates", it was previously not clear how to formalize them in a logic like HOL, where "termination" has no direct correspondence.

Fortunately, our framework provides adequate tools to express such notions, since termination is modeled by membership in the domain. Given the nested zero function Z from in §2.1.1, we can state and prove that Z returns zero whenever it terminates:

lemma *Z-zero*: $x \in dom_Z \Longrightarrow Z\ x = 0$
by (*induct rule*: $pinduct_Z$) *auto*

The proof is just as simple as if we already knew that the function is total: Induction and simplification, but using the partial induction and simplification rules. Then termination of Z is equally simple, using mathematical induction and the domain introduction rules, making use of the lemma to show termination of the outer recursive call:

lemma *Z-terminates*: $x \in dom_Z$
proof (*induct x rule*: *less-induct*)
  fix x **assume** *IH*: $\bigwedge y.\ y < x \Longrightarrow y \in dom_Z$
  show $x \in dom_Z$
  proof *cases*
    assume $x = 0$ **thus** $x \in dom_Z$ **by** (*auto intro*: dom_Z*-intro*)
  next
    assume $x \neq 0$
    with *IH* **have** $(x - 1) \in dom_Z$ **by** *auto*
    hence $Z\ (x - 1) = 0$ **by** (*rule Z-zero*)
    hence $Z\ (x - 1) \in dom_Z$ **by** (*auto intro*: dom_Z*-intro*)

[2] To compare wellfounded induction with functional induction, it is an interesting exercise to add even more nesting to the nested-zero example by changing the definition to $Z\ n = (\textit{if}\ n = 0\ \textit{then}\ 0\ \textit{else}\ Z\ (Z\ (Z\ (Z\ (n - 1)))))$, and then trying to prove the lemma $\forall n.\ Z\ n = 0$ once by *nat*-induction, where the property can be assumed on smaller arguments and once by functional induction, where the property can be assumed on the arguments of all recursive calls.

  from ‹$(x - 1) \in \mathit{dom}_Z$› **and** ‹$Z\ (x - 1) \in \mathit{dom}_Z$›
  show $x \in \mathit{dom}_Z$ **by** (*auto intro*: dom_Z-*intro*)
 qed
qed

This shows that our aproach separates the partial correctness proof (Z returns zero if it terminates) and the termination proof (Z terminates), which makes reasoning very natural. In §2.7, we give more examples of nested recursions.

2.4.1 A termination rule for nested recursion

We now have another look at the termination rule from §2.3.2. For the nested-zero function Z, we get the rule

$$\frac{\mathit{wf}\ R \qquad \bigwedge n.\ n \neq 0 \Longrightarrow (n - 1, n) \in R \qquad \bigwedge n.\ n \neq 0 \Longrightarrow (Z\ (n - 1), n) \in R}{\forall x.\ x \in \mathit{dom}_Z}$$

Now the second termination condition involves the function Z. We would like to use the lemma $x \in \mathit{dom}_Z \Longrightarrow Z\ x = 0$ to prove that the outer call is decreasing. But with this rule, we cannot do this, since we cannot assume termination of the inner call.

The following variant of the termination rule solves the problem:

$$\frac{\mathit{wf}\ R \qquad \bigwedge n.\ n \neq 0 \Longrightarrow (n - 1, n) \in R \qquad \bigwedge n.\ n \neq 0 \Longrightarrow \boxed{n - 1 \in \mathit{dom}_Z} \Longrightarrow (Z\ (n - 1), n) \in R}{\forall x.\ x \in \mathit{dom}_Z}$$

By the new assumption, marked with a box, we can now use the fact that the inner call terminates for proving that the outer call is decreasing. This is just enough to apply the lemma and conclude that the outer call is decreasing, since we are just left with the trivial goal $n \neq 0 \Longrightarrow 0 < n$.

This new termination rule reduces the termination proof for Z to

termination Z
by (*relation less-than*) (*auto simp only*: *Z-zero*)

2.4.2 Proving the nested termination rule

The system proves the nested termination rule automatically by wellfounded induction on the relation R. In the body of the induction we need to prove $x \in \mathit{dom}_f$ under the assumption $\bigwedge z.\ (z, x) \in R \Longrightarrow z \in \mathit{dom}_f$. Using the introduction rule for dom_f, it is sufficient to show $\Gamma_i \Longrightarrow r_i \in \mathit{dom}_f$ for each recursive call. Now we proceed from the innermost call to the outer calls. By assumption, we have for the innermost call $\Gamma_1 \Longrightarrow (r_1, x) \in R$ and by induction hypothesis we have $\Gamma_1 \Longrightarrow r_1 \in \mathit{dom}_f$. Now we can use this fact to strengthen the assumptions for the outer recursive calls, and proceed in the same way. The outermost calls come last, and for them we can assume termination of all inner calls.

2.5 Extraction of Recursive Calls and Congruence Rules

The definition process described in §2.2 assumed an algorithm to extract the recursive calls from the right-hand side of an equation. In this section, we describe the extraction process, which takes a term and produces a set of recursive calls $[\Gamma_1 \leadsto r_1]$, …, $[\Gamma_k \leadsto r_k]$ such that the associated congruence condition is provable.

For the first-order case, such an extraction was already given by Boyer and Moore [23], but, as we have seen, higher-order recursion produces some difficulties, which were solved by Slind [98], who invented a generic extraction procedure that is parametrized by congruence rules. Our extraction is essentially the same, and our main motivation for presenting it here is to make this thesis more self-contained.

Note that the extraction of calls critically influences the definition of the domain of the function and hence the termination proof obligations and the induction principle that the definition produces. Hence it is sometimes important for users to develop a basic understanding of this process.

2.5.1 Congruence rules

Congruence rules are used in contextual rewriting to accumulate context from the structure of a term. For example, when rewriting the then-part of an if-then-else expression, we may use the condition as a local assumption. In the *else* part, we may assume its negation.

This knowledge, which exploits a property of *if*, is not hardcoded in the rewriter, but expressed by a congruence rule:

$$\frac{c = c' \qquad c' \Longrightarrow t = t' \qquad \neg\ c' \Longrightarrow e = e'}{(\textsf{\textit{if}}\ c\ \textsf{\textit{then}}\ t\ \textsf{\textit{else}}\ e) = (\textsf{\textit{if}}\ c'\ \textsf{\textit{then}}\ t'\ \textsf{\textit{else}}\ e')}\ (\textit{if-cong})$$

In its very characteristic form, the congruence rule can be interpreted as a recipe for rewriting an expression of the form $\textsf{\textit{if}}\ c\ \textsf{\textit{then}}\ t\ \textsf{\textit{else}}\ e$: First, rewrite the condition to some c', then rewrite the ***then*** and ***else*** part, under the assumption c' or $\neg\ c'$, respectively.

Similar congruence rules exist for control structures like *case* or *let*, for higher-order combinators like *map* and *filter*, and also for bounded quantifiers like $\forall \cdot{\in}\cdot$. The congruence rules tell the contextual rewriter how to extend the context when traversing the term. Figure 2.1 shows congruence rules for other commonly used constructs. It is worth noting that some of the rules not only introduce new assumptions in the context but also bind new variables. For example, the rule for *map* introduces a variable x, which is constrained to be an element of the list.

2.5.2 Extracting calls

The extraction can now be described as a recursive algorithm that traverses a term t and incrementally builds a context, starting from the empty context:

1. If t does not contain f, then there are no more calls and we can stop.

$$\frac{xs = ys \qquad \bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x}{map\ f\ xs = map\ g\ ys}\ (map\text{-}cong) \qquad \frac{M = N \qquad \bigwedge x.\ x = N \Longrightarrow f\ x = g\ x}{Let\ M\ f = Let\ N\ g}\ (let\text{-}cong)$$

$$\frac{A = B \qquad \bigwedge x.\ x \in B \Longrightarrow P\ x = Q\ x}{(\forall x{\in}A.\ P\ x) = (\forall x{\in}B.\ Q\ x)}\ (Ball\text{-}cong) \qquad \frac{P = P' \qquad P' \Longrightarrow Q = Q'}{(P \wedge Q) = (P' \wedge Q')}\ (conj\text{-}cong)$$

$$\frac{f = g \qquad x = y}{f\ x = g\ y}\ (app\text{-}cong) \qquad \frac{\bigwedge x.\ f\ x = g\ x}{(\lambda x.\ f\ x) = (\lambda x.\ g\ x)}\ (lam\text{-}cong)$$

$$\frac{x = y \qquad y = 0 \Longrightarrow a = b \qquad \bigwedge n.\ y = Suc\ n \Longrightarrow f\ n = g\ n}{(\mathsf{case}\ x\ \mathsf{of}\ 0 \Rightarrow a \mid Suc\ n \Rightarrow f\ n) = (\mathsf{case}\ y\ \mathsf{of}\ 0 \Rightarrow b \mid Suc\ n \Rightarrow g\ n)}\ (nat\text{-}case\text{-}cong)$$

Figure 2.1: Congruence rules for commonly used constants

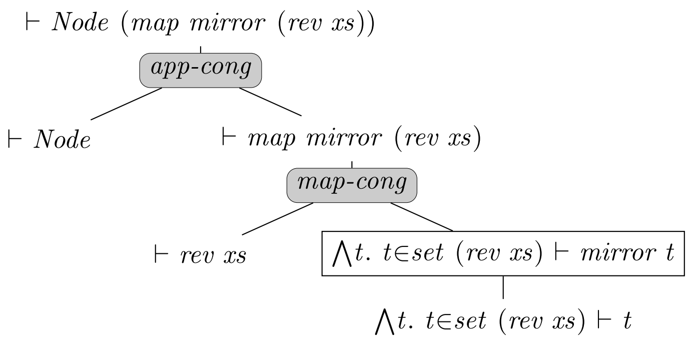
2. If t has the form $f\ r$, then we have the recursive call $[\Gamma \leadsto r]$. Continue with the extraction on subterm r, in the same context.

3. Otherwise, try if any of the congruence rules can be applied. If so, then we have one branch for each premise in the congruence rule. The current context Γ is extended with the context from that premise.

The last two congruence rules that are tried are always *app-cong* and *lam-cong*. This ensures that the last case always works: Any application is just split into two parts by *app-cong*, and if a term still contains *f*, but is not applied, then we can apply *lam-cong*.

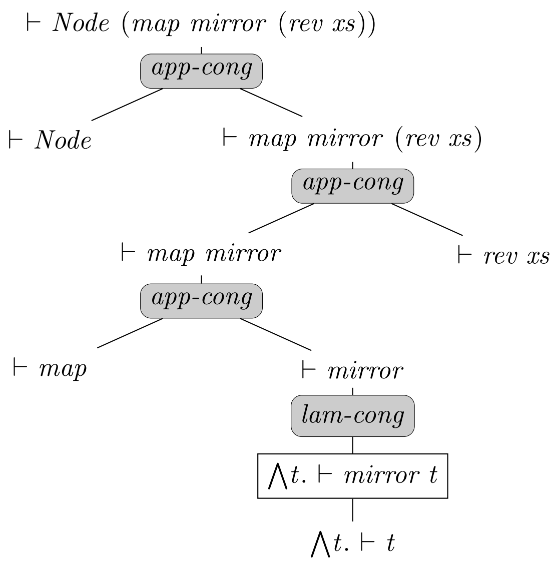
The extraction algorithm can be seen as a process of contextual rewriting to prove the congruence condition. Congruence rules are used to build up the right context for the recursive calls.

Example 2.2. We reconsider the function *mirror* given in §2.2. The extraction process is illustrated by a tree. Each node consists of a context and a term, written $\Gamma \vdash t$. At the root we have the complete right-hand side of the equation and the empty context. Then the term is split into components via congruence rules:



Since there is no special congruence rule for the *Node* constructor, the *app-cong* rule is applied, and it simply splits the application in two parts. The *Node* constructor on the left-hand side is uninteresting, since there is no recursive call here. On the right hand side, we have *map mirror* (*rev xs*), which now matches the *map-cong* rule. Following the structure of that rule, we get two branches, one for *rev xs*, which is again uninteresting, and one for *mirror t*, which now appears in a context extended by a new variable t and an assumption $t \in \mathit{set}\ (\mathit{rev}\ \mathit{xs})$. At this point, we have found a recursive call, since *mirror* is now fully applied. We continue this search on the subterm t, where it immediately terminates, since there are no more occurrences of *mirror*.

If the rule *map-cong* were not present, we would still get a tree, but it would instead look like this:



Now the extracted recursive call would be $[\bigwedge x.\ \leadsto x]$, which is not helpful, as we have seen.

This example demonstrates that the extraction mechanism depends on the configuration via congruence rules, which encode instructions for dealing with higher-order constructs. This has the advantage that it makes the package very flexible. The disadvantage however is that users sometimes need to know how the extraction works, in order to feed it the right congruence rules.

The result of the extraction process is a set of calls $[\Gamma_1 \leadsto r_1], \ldots, [\Gamma_k \leadsto r_k]$. This set is used in the definition process as we have seen in §2.2. In particular it determines the definition of the graph and the domain and, as a consequence, the form of the induction rule and the termination proof obligations.

The way the calls are constructed guarantees that the congruence condition given in §2.2 is provable automatically. The straightforward proof simply follows the tree structure above.

2.5.3 Congruence rules and evaluation order

Higher-order logic differs from functional programming languages in that it has no built-in notion of evaluation order. A program is just a set of equations, and it is not specified how they must be evaluated. However, when reasoning about termination of recursive functions, an implicit notion of evaluation order sneaks in. The evaluation order is specified by the congruence rules we are using. For example, consider the following simple recursion on natural numbers:

$$f\ n = (n = 0 \lor f\ (n - 1))$$

Whether this is a total function or not depends on how we interpret $\lor$. We could use the semantics known from ML, where *orelse* and *andalso* are strict in the left argument but non-strict in the right one. Then the function terminates because $n \neq 0$ implies $n - 1 < n$. However, if the disjunction is strict in both arguments we get nontermination. (Recall that on natural numbers, $n - 1$ may be equal to n if n is zero.)

HOL itself does not make this distinction, since there is no explicit notion of undefinedness. Instead, the congruence rules that we use to extract the recursive calls will determine which function we get out. Without any congruence rules, the extraction will regard disjunction as strict in both arguments, and our function has the empty domain. However, we can give a congruence rule for disjunction that gives it the behaviour known from ML:

$$\frac{P = P' \qquad \neg\ P' \Longrightarrow Q = Q'}{(P \lor Q) = (P' \lor Q')}\ (\textit{disj-cong})$$

Now the definition of f above gives us the total function we expect. The termination proof will use the extra condition that we obtained from the congruence rule.

However, as evaluation is not a hard-wired concept, we could just turn everything around by declaring a different congruence rule:

$$\frac{\neg\ Q' \Longrightarrow P = P' \qquad Q = Q'}{(P \lor Q) = (P' \lor Q')}\ (\textit{disj-cong2})$$

This would allow us to make the reverse definition:

$$f\ n = (f\ (n - 1) \lor n = 0)$$

This already shows that the congruence rules that we need might depend on the function we are defining. Note that the meaning of disjunction does not change.

One could argue that *disj-cong2* is unnatural, and that *disj-cong* should be enabled by default. Adding this rule will always make the termination proof simpler, since the recursive calls are restricted by an extra condition. However, as another consequence we get a weaker induction rule. Consider for example a function that checks if some element in a list satisfies some (fixed) predicate *test*:

$$\begin{aligned} &\textit{testany}\ [\,] &&= \textit{False} \\ &\textit{testany}\ (x{:}xs) &&= \textit{test}\ x \lor \textit{testany}\ xs \end{aligned}$$

Obviously, *testany* terminates just by structural recursion over the list, so *disj-cong* is not needed here. If we still add it, the function is not changed, but in the induction rule, the inductive hypothesis is now guarded by a condition:

$$\frac{P\ [\,] \qquad \bigwedge x\ xs.\ (\neg\ \textit{test}\ x \Longrightarrow P\ xs) \Longrightarrow P\ (x{:}xs)}{P\ a}$$

When we do induction with this rule, we will always have to show $\neg$ *test x* before we can apply the induction hypothesis. So in this case we get a better induction principle by avoiding the unnecessary congruence rule. Of course, for this example we can just use standard list induction, but in other situations the custom induction rule might be important.

These examples show that, in general, there is no "best" or "complete" set of congruence rules. The default setup in Isabelle is rather conservative, relying on the user to manually add rules when needed. Nonetheless, the basic set of predefined congruence rules often proves sufficient.

2.6 Extensions

The core recursion infrastructure described above is already quite powerful. In this section we describe some useful extensions: default values, tail recursion, pattern matching, mutual recursion and currying.

2.6.1 Default values

Recall that we model partial functions as underspecified total functions. Outside their domain, we cannot determine their value. Sometimes this is not desirable, as there may be a natural completion of the function that better suits the needs of the application. For example, a function that returns values of an *option* type coud be completed to return *None* in such cases.

Here, we note the difference between the *algorithm* that is specified by the recrsive equations and the *function* that we define in the logic. While the algorithm cannot return anything when it does not terminate, the function actually has a value, and we can specify that value (which we call the *default value*) at definition time.

The only change that is needed is to replace the description operator *THE* which we used to define the function (cf. §2.2.2) by a variant that takes a default value:

$$\begin{aligned}&\mathit{THE\text{-}default} \;::\; \alpha \Rightarrow (\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha\\ &\mathit{THE\text{-}default}\ d\ P = (\textsf{if}\ \exists!x.\ P\ x\ \textsf{then}\ \mathit{THE}\ x.\ P\ x\ \textsf{else}\ d)\end{aligned}$$

Then we can define functions with a user-specified default value d, which may even depend on x. We can then derive the following theorem:

$$x \notin \mathit{dom} \Longrightarrow f\,x = d\ x$$

To motivate the use of default values, consider a function that checks some kind of certificate:

$$\mathit{checker} \;::\; \mathit{cert} \Rightarrow \mathit{bool}$$

The implementation of *checker* may be a very complicated algorithm, for which we can only prove partial correctness. Hence we have a theorem

$$c \in \mathit{dom}_{\mathit{checker}} \Longrightarrow \mathit{checker}\ c \Longrightarrow P\ c$$

for some interesting property P.

Now, if we can define *checker* such that it returns *False* when given a value that is not in the domain, then we can remove the domain condition from the above theorem, which can make subsequent reasoning simpler:

$$\mathit{checker}\ c \Longrightarrow P\ c$$

Note that default values are just a logical concept and have no operational meaning. If the function *checker* is run on something outside its domain, it will still loop instead of returning *False*. However, we have a theorem that it is equal to *False*, logically. Here the difference between the logical view and the algorithmic view of recursive function becomes very apparent.

2.6.2 Tail recursion

The partial simplification rules generated by the function package are guarded by domain conditions. If the function does not always terminate, it is usually not possible to remove them (recall the example $U\ x = U\ x + \mathit{1}$, which is obviously inconsistent). However, there is an important special case for which unguarded recursion equations are derivable even for partial functions. This is the case when the function is tail-recursive, a fact that was first noticed and exploited by Manolios and Moore [68]. In HOL, tail-recursive functions could previously be defined by instantiating a *while* combinator, but that was a tedious manual process.

While it would be possible to automate the definition of tail-recursive functions using a *while* combinator, it turns out that we can achieve the same effect with the definition framework presented here, by deriving the unconstrained simplification rules afterwards. For this we use the default value feature described above, and give the function an arbitrary default value d that is independent from the input.

Now, the unconstrained recursion equation $f\ x = F\ f\ x$ can be proved as follows: For $x \in \mathit{dom}$, we just need to apply the partial simplification rule. Consider the case $x \notin \mathit{dom}$. By tail recursion we know that $F\ f\ x = f\ (g\ x)$, for some expression g. Now $g\ x$ cannot be in dom, since otherwise x would also be in dom. Since both x and $g\ x$ are not in dom, we have $f\ x = d = f\ (g\ x) = F\ f\ x$ which is our recursive equation.

This reasoning can be automated, and thus we can provide unconstrained recursion equations for the user, if the function is tail-recursive.

One important motivation for removing the domain conditions even for partial functions is that Isabelle/HOL's code generator can only handle unconditional equations. When we are able to derive them as theorems, then we can use all the existing code generation facilities [13, 48] to convert our Isabelle/HOL specifications to ML or Haskell programs. Note that this translation only preserves partial correctness, as the resulting code may be nonterminating.

2.6.3 Pattern matching

An extension of high practical importance is the support for definitions with pattern matching, where functions are not specified by a single equation $f\ x$

$= F\ f\ x$, but in terms of multiple equations with patterns. This extension is complex enough that we describe it in a chapter of its own (ch. 4).

2.6.4 Mutual recursion and currying

Our package implements mutual recursion by first reducing it to simple recursion on a suitable sum type. This is a very simple reduction, and it is already described by Slind [98], so we just give a small example: The functions *even* and *odd* with the equations

$$\begin{array}{lcl} \textit{even } 0 & = & \textit{True} \\ \textit{even } (\textit{Suc } n) & = & \textit{odd } n \\ \textit{odd } 0 & = & \textit{False} \\ \textit{odd } (\textit{Suc } n) & = & \textit{even } n \end{array}$$

are reduced to a single function *even-odd* $::$ $\textit{nat} + \textit{nat} \Rightarrow \textit{bool}$ with the equations

$$\begin{array}{lcl} \textit{even-odd } (\textit{Inl } 0) & = & \textit{True} \\ \textit{even-odd } (\textit{Inl } (\textit{Suc } n)) & = & \textit{even-odd } (\textit{Inr } n) \\ \textit{even-odd } (\textit{Inr } 0) & = & \textit{False} \\ \textit{even-odd } (\textit{Inr } (\textit{Suc } n)) & = & \textit{even-odd } (\textit{Inl } n) \end{array}$$

Then the individual functions are defined as

$$\begin{array}{lcl} \textit{even } n & = & \textit{even-odd } (\textit{Inl } n) \\ \textit{odd } n & = & \textit{even-odd } (\textit{Inl } n) \end{array}$$

and we easily derive the original recursive equations from this.

We must also produce an appropriately modified induction principle. The (total) induction rule for *even* and *odd* involves two induction predicates P and Q:

$$\frac{\bigwedge n.\ Q\ n \Longrightarrow P\ (\textit{Suc } n) \qquad \begin{array}{c} P\ 0 \\ Q\ 0 \end{array} \qquad \bigwedge n.\ P\ n \Longrightarrow Q\ (\textit{Suc } n)}{P\ a \wedge Q\ a}$$

We can handle currying in a similar way: If a function has multiple arguments, we first define the corresponding uncurried function which takes a tuple. From that function we then define the curried function and derive the equations.

These transformations can be used as wrappers around the core of the package. They reduce currying and mutual recursion to simple functions, such that the rest of the definition infratsructure just needs to handle a single function with one argument.

2.7 Further Examples

In this section, we want to present some more examples that demonstrate how the partial induction rule simplifies reasoning about nested recursive and partial functions. Some of these examples use pattern matching, which we have not discussed yet. It will be the subject of Chapter 4.

2.7.1 Nested list reversal

The following function defines list reversal in an uncommon way that does not need any auxiliary function or additional parameter. It was given to the author as a challenge problem by Konrad Slind, and it is probably more of a puzzle than a sensible implementation.

$$\begin{aligned}&\mathit{Rev}\ [\,] &&= [\,]\\ &\mathit{Rev}\ (x{:}xs) &&= \mathsf{case}\ \mathit{Rev}\ xs\ \mathsf{of}\ [\,] \Rightarrow [x]\ \mid\ y{:}ys \Rightarrow y{:}\mathit{Rev}\ (x{:}\mathit{Rev}\ ys)\end{aligned}$$

In fact, proving that *Rev* is equivalent to Isabelle's built-in function *rev* is just a straightforward functional induction. But the nesting made this function hard to define and reason about with previous tools. Using the partial induction rule, we can jump forward and prove that *Rev* is equal to *rev* on its domain:

lemma *Rev-eq-rev*[*simp*]: $xs \in dom_{Rev} \Longrightarrow Rev\ xs = rev\ xs$
by (*induct xs rule*: $pinduct_{Rev}$) (*auto split*: *list.splits*)

This is so easy because the induction rule is tailored to the recursive structure of the function. Now, using this lemma, termination is not hard to prove, since much about *rev* is already known (in particular, *length* (*rev xs*) = *length xs*). The termination order can be found automatically, so we just invoke the automated termination prover, which will be presented in Chapter 3:

termination by *lexicographic-order*

2.7.2 McCarthy's 91 function

The ninety-one function is a well-known challenge problem due to John McCarthy:

$$f91\ n = (\mathsf{if}\ 100 < n\ \mathsf{then}\ n - 10\ \mathsf{else}\ f91\ (f91\ (n + 11)))$$

The termination argument relies on the following lemma:

lemma *f91-estimate*:
  $n \in dom_{f91} \Longrightarrow n < f91\ n + 11$
by (*induct rule*: $pinduct_{f91}$) *auto*

Now we can proceed with a manual termination proof, which we give in Isar notation. Note how the assumption $n + 11 \in dom_{f91}$ is used for the outer call to discharge the hypothesis of the lemma:

termination
proof
  let $?R = measure\ (\lambda x.\ 101 - x)$ — The termination relation used
  show $wf\ ?R$ **..**

  fix $n :: nat$ **assume** $\neg\ 100 < n$ — Assumptions for both calls

  thus $(n + 11,\ n) \in ?R$ **by** *simp* — Inner call

  assume *inner-trm*: $(n + 11) \in dom_{f91}$ — Outer call
  with *f91-estimate* **have** $n + 11 < f91\ (n + 11) + 11$ **.**
  with $\langle\neg\ 100 < n\rangle$ **show** $(f91\ (n + 11),\ n) \in ?R$ **by** *simp*
qed

2.7.3 First order unification

A standard example of nested recursion is a unification algorithm on a simple first-order term language. This example was already presented by Slind [99], who in turn adapted it from Manna and Waldinger [67]. There are also other formalizations of unification, e.g. by Paulson [88]. Our claim is that we are the first to present a framework where the definition of the function and the reasoning about it are really natural, and not obfuscated by restrictions imposed by the tools. In the following description we focus on this aspect and do not discuss the formalization in detail. The complete theory can be found in the Isabelle 2009 distribution.

In our formalization, terms can be variables, constants and applications of one term to another:

$$
\begin{array}{l}
\textbf{datatype}\ \alpha\ \mathit{trm} = \\
\quad \mathit{Var}\ \alpha \\
\mid \mathit{Const}\ \alpha \\
\mid \mathit{App}\ (\alpha\ \mathit{trm})\ (\alpha\ \mathit{trm}) \quad (\textbf{infix} \cdot \mathit{60})
\end{array}
$$

Substitutions are modeled as association lists mapping variables to terms. The (parallel) application of a substitution σ to a term t is written $t \lhd s$, and we omit its straightforward definition. Composition of substitutions is written $\sigma_2 \circ \sigma_1$.

The unification function has the following definition:

$$
\begin{array}{l}
\mathit{unify}\ (\mathit{Const}\ c)\ (M \cdot N) = \mathit{None} \\
\mathit{unify}\ (M \cdot N)\ (\mathit{Const}\ c) = \mathit{None} \\
\mathit{unify}\ (\mathit{Const}\ c)\ (\mathit{Var}\ v) = \mathit{Some}\ [(v,\ \mathit{Const}\ c)] \\
\mathit{unify}\ (M \cdot N)\ (\mathit{Var}\ v) = \\
(\mathsf{if}\ \mathit{occ}\ (\mathit{Var}\ v)\ (M \cdot N)\ \mathsf{then}\ \mathit{None}\ \mathsf{else}\ \mathit{Some}\ [(v,\ M \cdot N)]) \\
\mathit{unify}\ (\mathit{Var}\ v)\ M = (\mathsf{if}\ \mathit{occ}\ (\mathit{Var}\ v)\ M\ \mathsf{then}\ \mathit{None}\ \mathsf{else}\ \mathit{Some}\ [(v,\ M)]) \\
\mathit{unify}\ (\mathit{Const}\ c)\ (\mathit{Const}\ d) = (\mathsf{if}\ c = d\ \mathsf{then}\ \mathit{Some}\ [\,]\ \mathsf{else}\ \mathit{None}) \\
\mathit{unify}\ (M \cdot N)\ (M' \cdot N') = \\
(\mathsf{case}\ \mathit{unify}\ M\ M'\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{None} \\
\ \mid \mathit{Some}\ \vartheta \Rightarrow \\
\qquad \mathsf{case}\ \mathit{unify}\ (N \lhd \vartheta)\ (N' \lhd \vartheta)\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{None} \\
\qquad \mid \mathit{Some}\ \sigma \Rightarrow \mathit{Some}\ (\vartheta \circ \sigma))
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{occ}\ u\ (\mathit{Var}\ v) & = & \mathit{False} \\
\mathit{occ}\ u\ (\mathit{Const}\ c) & = & \mathit{False} \\
\mathit{occ}\ u\ (M \cdot N) & = & u = M \lor u = N \lor \mathit{occ}\ u\ M \lor \mathit{occ}\ u\ N
\end{array}
$$

Note that the nesting of the recursion is not directly of the form *unify* (… *unify*(…) …). Instead the calls are connected via their contexts: The assumption $\mathit{unify}\ M\ M' = \mathit{Some}\ \vartheta$ occurs in the context of the second call.

To demonstrate how the partial induction rule simplifies reasoning, we prove partial correctness of the unification algorithm first:

$$
\frac{(M,\ N) \in \mathit{dom}_{\mathit{unify}} \qquad \mathit{unify}\ M\ N = \mathit{Some}\ \sigma}{\mathit{MGU}\ \sigma\ M\ N}
$$

The proof is by induction using the partial induction rule *unify.pinduct*. Figure 2.2 shows the full proof.

```
lemma unify-partial-correctness:
  (M, N) ∈ dom_unify ⟹ unify M N = Some σ ⟹ MGU σ M N
proof (induct M N arbitrary: σ rule: pinduct_unify)
  case (7 M N M' N' σ) — The interesting case

  then obtain ϑ1 ϑ2
    where unify M M' = Some ϑ1 and unify (N ◁ ϑ1) (N' ◁ ϑ1) = Some ϑ2
    and [simp]: σ = ϑ1 ∘ ϑ2
    and MGU-inner: MGU ϑ1 M M'
    and MGU-outer: MGU ϑ2 (N ◁ ϑ1) (N' ◁ ϑ1)
    by (auto split: option.split-asm)

  show MGU σ (M · N) (M' · N')
  proof — We have a unifier:
    from MGU-inner and MGU-outer
    have M ◁ ϑ1 = M' ◁ ϑ1 and N ◁ ϑ1 ◁ ϑ2 = N' ◁ ϑ1 ◁ ϑ2
      by (auto simp: MGU-def Unifier-def)
    thus M · N ◁ σ = M' · N' ◁ σ by simp
  next — The unifier is most general:
    fix σ' assume M · N ◁ σ' = M' · N' ◁ σ'
    hence M ◁ σ' = M' ◁ σ' and Ns: N ◁ σ' = N' ◁ σ' by auto

    with MGU-inner obtain δ where eqv: σ' =_s ϑ1 ∘ δ
      by (auto simp: MGU-def Unifier-def)

    from Ns have N ◁ ϑ1 ◁ δ = N' ◁ ϑ1 ◁ δ
      by (simp add: eqv-dest[OF eqv])

    with MGU-outer obtain ϱ where eqv2: δ =_s ϑ2 ∘ ϱ
      by (auto simp: MGU-def Unifier-def)

    have σ' =_s σ ∘ ϱ
      by (rule eqv-intro, auto simp: eqv-dest[OF eqv] eqv-dest[OF eqv2])
    thus ∃δ. σ' =_s σ ∘ δ ..
  qed
qed (auto split: split-if-asm) — Solve the remaining cases automatically
```

Figure 2.2: Partial correctness of *unify*

In order to prove termination of the function, we need to establish two properties which have to do with occurrences of variables. First, substitutions produced by *unify* never introduce new variables (we omit the trivial definition of *vars-of*):

$$\frac{(M,\ N) \in dom_{unify} \qquad unify\ M\ N = Some\ \sigma}{vars\text{-}of\ (t \lhd \sigma) \subseteq vars\text{-}of\ M \cup vars\text{-}of\ N \cup vars\text{-}of\ t}$$

Second, if *unify* returns a substitution σ, then σ is either the identity substitution ($=_s$ denotes equivalence of substitutions) or it eliminates a variable v, which means that for any term t, $t \lhd \sigma$ no longer contains v.

$$\frac{(M,\ N) \in dom_{unify} \qquad unify\ M\ N = Some\ \sigma}{(\exists\, v{\in}vars\text{-}of\ M \cup vars\text{-}of\ N.\ \forall\, t.\ v \notin vars\text{-}of\ (t \lhd \sigma)) \lor \sigma =_s [\,]}$$

These lemmas are again proved by partial induction, where the recursive case is the interesting one. Again, the partiality has no influence on the structure of the induction proof.

The termination proof then merely puts these results together, using the lexicographic combination of the measures $\lambda(M,\ N).\ card\ (vars\text{-}of\ M \cup vars\text{-}of\ N)$ and $\lambda(M,\ N).\ size\ M$. We then get total correctness as a corollary, since we now know that $(M,\ N) \in dom_{unify}$ always holds.

2.7.4 Depth-first search

In ongoing work, Berghofer and Reiter are formalizing various constructions on finite automata. Their formalization also includes an abstract specification of depth-first search in directed graphs, which is developed in an axiomatic context (using Isabelle's locale mechanism [8]), such that it can later be instantiated to different representations of the graph that is searched and the data structure that collects the results. The depth-first search formalization is loosely based on a previous formalization by Nishihara and Minamide [75, 82]. A similar formalization in HOL4 is given by Owens and Slind [84].

The graph is modelled abstractly by a type *node* and the functions

succs	$::\ node \Rightarrow node\ list$	(successor nodes)
is-node	$::\ node \Rightarrow bool$	(wellformedness predicate)

Not every value of type *node* is necessarily a node in the graph, and the predicate *is-node* models the set of valid nodes, which must be finite. For a valid node, the function *succs* returns the list of successors.

During the traversal, nodes are collected in a data structure of another abstract type C, which behaves like a set of nodes:

empt	$::\ C$	(empty collection)
ins	$::\ node \Rightarrow C \Rightarrow C$	(insert operation)
memb	$::\ node \Rightarrow C \Rightarrow bool$	(membership test)
invariant	$::\ C \Rightarrow bool$	(collection invariant)

The axioms that describe these operations are given in Fig. 2.3. The advantage of working with an abstract specification of graphs and collections is that the algorithm can be instantiated later with concrete data structures suited for the particular applications, e.g., matrices and BDDs.

The depth-first search function is defined as follows:

$$\textit{is-node}\ x \Longrightarrow \textit{is-node}\ y \Longrightarrow \textit{invariant}\ S \Longrightarrow \neg\ \textit{memb}\ y\ S$$
$$\Longrightarrow \textit{memb}\ x\ (\textit{ins}\ y\ S) = (x = y \lor \textit{memb}\ x\ S)$$

$$\textit{is-node}\ x \Longrightarrow \neg\ \textit{memb}\ x\ \textit{empt}$$

$$\textit{is-node}\ x \Longrightarrow \forall\, y{\in}\textit{set}\ (\textit{succs}\ x).\ \textit{is-node}\ y$$

$$\textit{invariant}\ \textit{empt}$$

$$\textit{is-node}\ x \Longrightarrow \textit{invariant}\ S \Longrightarrow \neg\ \textit{memb}\ x\ S \Longrightarrow \textit{invariant}\ (\textit{ins}\ x\ S)$$

$$\textit{finite}\ \textit{is-node}$$

Figure 2.3: Axiomatic context for the depth-first search algorithm

$$\textit{dfs} :: C \Rightarrow \textit{node list} \Rightarrow C$$

$$\textit{dfs}\ S\ [\,] = S$$
$$\textit{dfs}\ S\ (x{:}xs) =$$
$$(\textsf{if}\ \textit{memb}\ x\ S\ \textsf{then}\ \textit{dfs}\ S\ xs\ \textsf{else}\ \textit{dfs}\ (\textit{ins}\ x\ S)\ (\textit{succs}\ x\ @\ xs))$$

Even though the axiomatization ensures that the graph is finite, *dfs* is a partial function, since the behaviour of *ins* and *succs* is only specified when the invariants *is-node* and *invariant* are satisfied. Otherwise they may return a value that leads to nontermination of *dfs*.

However, termination of *dfs S xs* can be proved if *invariant S* and $\forall\, x{\in}xs.$ *is-node x* hold. Moreover, tail-recursion ensures that unconditional equations can be generated.

2.7.5 Pseudo-division for multivariate polynomials

As another example of a partial function, we refer to Chaieb's formalization of multivariate polynomials [28]. That formalization includes a definition of pseudo-division, which is an inherently partial operation, since it only terminates if the polynomials are normalized. Using our definition facilities, Chaieb was able to introduce the pseudo-division function, and prove the relevant properties. Moreover, since the function is tail recursive, it can be executed and used as part of a reflected decision procedure.

2.8 Limitations

In this section, we briefly discuss some general limitations of our package.

2.8.1 Higher-order nesting

We have seen how our package gracefully handles higher-order and nested recursion. However, by combining the two, we can take the difficulty to a new level and make the automation fail. Here is a simple example — yet another silly way of defining the constant zero function:

$$\textit{zero}\ n = \textit{fun-pow}\ n\ \textit{zero}\ 0$$

It is easy to see that $\mathit{zero}\ 0 = \mathit{id}\ 0 = 0$ and hence $\mathit{zero}\ n = \mathit{zero}\ (\mathit{zero}\ (\ldots\ (\mathit{zero}\ 0)\ \ldots) = 0$. The problem is that we cannot give a useful congruence rule for *fun-pow*, the function exponentiation. Intuitively, we would like to express that in *fun-pow n f x*, the function f is called on the values *fun-pow i f x* for all $i < n$. But this contains the very same pattern again, which makes the extraction of recursive calls loop.

This example shows that the extraction of recursive calls using congruence rules is just an approximation that works well in practice but may also fail.

We can circumvent this problem by expanding the higher-order recursion into a mutual recursion by adding the recursion equations for *fun-pow* to the definition of *zero*:

$$\mathit{zero}\ n = \mathit{zeropow}\ n\ 0$$

$$\mathit{zeropow}\ 0\ x = x$$
$$\mathit{zeropow}\ (\mathit{Suc}\ n)\ x = \mathit{zero}\ (\mathit{zeropow}\ n\ x)$$

Then we no longer need a congruence rule and can proceed in the normal way to prove termination of the mutual and nested recursion.

2.8.2 Undefinedness does not propagate

If we expect that the domain dom_f models the set of values where of f terminates, it can be a little surprising to see that for the function

$$g\ x = U\ x$$

the associated domain dom_g is the universal set, although g calls U, whose domain is empty. The reason is that dom_g arises from the analysis of the recursion in the definition of g, and a non-recursive function always terminates in our sense.

It is possible to change the analysis to a more global one, where the domain of g would also depend on the domain of U. However, recall that the we introduced the domain primarily to simplify partial correctness proofs and not as a faithful model of termination with respect to some evaluation mechanism. Note that to obtain the latter, we would also have to settle on a fixed evaluation order and a fixed set of congruence rules.

It seems that the practical benefit of getting stronger induction and simplification rules outweighs the somewhat unintuitive property that undefinedness does not propagate.

2.8.3 Other forms of recursion

There are other forms of recursive definitions that are not based on wellfounded recursion. One example is corecursion, where the output of a function is a coinductive datatype, which can be infinite.

For example, the function

$$\mathit{from}\ n = n{:}\mathit{from}\ (\mathit{Suc}\ n)$$

defines an infinite sequence of numbers. Note that this can only work for coinductive lists, not for normal inductive ones, which are finite by construction.

Wellfounded recursion cannot define functions like *from*, and other tools would be required to introduce them.

2.9 Related Work

General recursion in proof assistants Generating termination conditions and induction schemes from recursive function definitions was first done by Boyer and Moore in NQTHM [23], the predecessor of ACL2 [60]. Today, ACL2 still works in essentially the same way: Functions must be proved total at definition time by giving the appropriate measure, which can sometimes be inferred by the system. As opposed to a definitional extension, recursion is built into the system itself and must be trusted. As ACL2 uses first-order logic, there is no higher-order recursion or congruence rules. ACL2 also supports the definition of (possibly partial) tail-recursive functions [68]. Then no termination proof is needed.

Both Isabelle and HOL4 [44] include (different versions of) the definitional recursion package TFL, a work by Slind [97, 98]. TFL supports the definition of total recursive functions by using the specialized fixed-point combinator *wfrec* and a wellfounded relation given by the user. Proving termination amounts to showing that the relation is wellfounded and recursive calls are decreasing. Optionally, termination arguments can be deferred by replacing the relation by its specification using a choice operator.

HOL Light [51] provides a similar mechanism, also based on a fixed-point combinator. Furthermore, by a clever combination with tail recursion, termination proof obligations only arise from non-tail calls, even if the function as a whole is not tail recursive. The drawback of this approach is that no induction principles can be generated. There is no general support for higher-order recursion.

In Coq [15], a recent package by Barthe, Forest, Pichardie, and Rusu [9] allows definitions in a manner similar to TFL. However, nested and higher-order recursion are not supported.

Partiality and domain predicates The idea of generating an explicit description of a function's domain emerged many times.

Finn, Fourman, and Longley [38] describe how partial functions can be axiomatized consistently in a total higher-order logic, by having their equations guarded by domain predicates. The domain predicates are again specified by recursive equations, but (as is justified semantically), a domain predicate for the domain predicate is not required.

Dubois and Donzeau-Gouge [34] replace the recursive domain with an inductive one, which makes the approach (in principle) amenable to implementation as a definitional extension to Coq. However, they did not provide an implementation.

Giesl [42] studies the use of functional induction for partial functions, and shows that it is applicable and useful. The partial induction rule is similar to the one we are using, and is proved sound with respect to the semantics of programs. As this employs a modified notion of truth for formulae, it is not directly applicable to higher-order logic, whose semantics is fixed. However, Giesl managed to extend existing induction provers with his calculus with little effort.

Bove and Capretta [19, 21] investigate how general recursion can be integrated into type theory, which by default only admits structural recursion. The idea of the approach is to produce an inductively defined domain predicate from the function specification, and then define the actual function by *structural* recursion on that inductive predicate. This approach also underlies the implementation of Coq's `Function` package [9]. The disadvantage is that there is no general type of partial recursive functions, but instead each function has its own private type of the form $\sigma \to dom_f\ \sigma \to \tau$. Either impredicativity or a coinductive construction [22] can be used to overcome this. Note that in our simply-typed framework this is not an issue, since partial functions are just underspecified total functions (of type $\sigma \to \tau$).

A different approach for dealing with non-termination is to work in a logic that features a "native" notion of partiality. One such logic is domain theory, where any computable function can easily be defined, since general fixed points exist. On the other hand, resoning in domain theory comes with a certain overhead, since induction is restricted to admissible predicates. This leads to additional admissibility and definedness proof obligations that can make reasoning harder.

Nested recursion Nested recursion is currently not well-supported in theorem provers. Slind's TFL package provides some support using a provisional induction rule [99], but the resulting proof obligations are clumsy.

The approach sketched by Dubois and Donzeau-Gouge [34] supports a user-specified *post condition* that can encode the property required for the termination proof of a nested recursive definition. Later, Krstić and Matthews [64] suggested a very similar notion they called *inductive invariant*. Inductive invariants are given by the user at definition time and are used to approximate the results of nested recursive call to make the termination proof work. However, no convenient reasoning principles are given for them, and one must resort to general wellfounded induction, which is more awkward to use than functional induction.

In our approach, such properties are simply expressed as ordinary lemmas about possibly partial functions, constrained by a domain predicate and provable by the partial induction rule. This follows Giesl's approach [41, 42], but it does not require a new notion of truth, since all constructions happen in standard HOL.

Bove and Capretta [20] can only support nested recursion by defining the domain and the function simultaneously, which requires the underlying theory to support simultaneous inductive-recursive definitions as described by Dybjer [35]. In contrast, our classical setting avoids this issue, since the domain is not required for the function definition but only introduced for the purpose of convenient reasoning.

Chapter 3

Termination Proofs

Contents

3.1 Introduction

In this chapter, we will develop automated techniques to solve the termination proof obligations that arise from function definitions.

There exists a large body of research on automating termination proofs, and many fully-automated provers have been developed. We will discuss the most important approaches in §3.2, but first we briefly point out some notable differences between the requirements that most existing termination provers were designed for and our specific setting.

Interactive Environment Although we are interested in automation, our theorem proving environment is interactive. This means that we should not just fail completely when we meet a problem that cannot be solved automatically, but be prepared to take hints from the user in some form, which help to continue the proof.

Furthermore, the termination problems that we are trying to solve are not isolated. While automated provers typically take the program as their only input and have to discover the proof that with no other information, our termination problems live in a rich theory, which may already provide many relevant lemmas. This is an advantage, because using such lemmas is certainly easier than rediscovering them and proving them on-the-fly. But we need a mechanism that allows feeding this information to the termination prover.

No operational semantics It is slightly peculiar that our language does not have a notion of evaluation. How are we to prove termination of something that does not run? The answer is that we must solve the wellfoundedness proof obligations arising in the previous chapter. Although it is helpful for our intuition that they correspond the termination of a function, they are just normal Isabelle goals that must be proved.

Conventional termination criteria are normally justified against the semantics of the programming language: They come with a soundness theorem that states that if the criterion is satisfied, then the program will eventually halt on every input. Since HOL programs are never actually run, we cannot even state such a theorem. Instead we already work with a more abstract description of programs: the call relation that we want to prove wellfounded.

Proof Generation Instead of using a criterion that is proved sound against some program semantics, we strictly adhere to the LCF approach by producing an explicit proof of the respective wellfoundedness property. It is therefore not sufficient that our method is sound, but it must produce a formal proof for each instance, which is checked by the Isabelle kernel. In his thesis, Chaieb discusses three different ways of accomplishing this [28], and all three variants (derived rules, reflection, certificates) will be used in this chapter.

As we have already seen for function definitions, the advantage of producing formal proofs is that an external (pen-and-paper) soundness proof is not necessary. As long as the system accepts the proofs produced by the tools, we can be sure that they are correct. Therefore, this chapter will contain very little meta-theoretic material.

3.1.1 Termination goals

The termination goals that we would like to solve have the following form:

$$\begin{array}{l} \mathit{0.}\ \mathit{wf}\ ?R \\ \mathit{1.}\ \bigwedge v_1 \ldots v_{m_1}.\ \Gamma_1 \Longrightarrow (r_1,\, l_1) \in\ ?R \\ \vdots \\ \mathit{n.}\ \bigwedge v_1 \ldots v_{m_n}.\ \Gamma_n \Longrightarrow (r_n,\, l_n) \in\ ?R \end{array}$$

The schematic variable *?R* denotes a wellfounded relation that needs to be supplied in the proof. The remaining subgoals simply state that the arguments of recursive calls must decrease with respect to *?R*.

We call this format of termination goal the *open format*. The open format is convenient for interactive proofs, which usually first instantiate *?R* with an explicit relation. Typically, proving the wellfoundedness and decrease condition is then straightforward.

However, for the purpose of automation, we prefer the *closed format*, which has no schematic variable and provides more flexibility for refinement and simplification:

$$\begin{aligned}&\mathit{wf}\ (\{\ (r_1,\ l_1) \mid v_1 \ldots v_{m_1}.\ \Gamma_1\ \}\\ &\quad \cup \ldots\\ &\quad \cup \{\ (r_n,\ l_n) \mid v_1 \ldots v_{m_n}.\ \Gamma_n\ \})\end{aligned}$$

Here, each recursive call is expressed by a relation comprehension, and we must prove that their union is wellfounded.

Both formats of termination goals are equivalent and it is easy to convert automatically between one and the other. Given an open goal, we convert it to a closed one by substituting the union of comprehensions for $?R$. Then the subgoals $1. - n.$ can be discharged, since they are true by construction. What remains is the wellfoundedness part: a closed goal. Conversely, we convert a closed goal to the open format using the rule $\mathit{wf}\ R \Longrightarrow S \subseteq R \Longrightarrow \mathit{wf}\ S$.

Having in mind where they originate from, we frequently refer to the individual relation comprehensions $\{\ (r_i,\ l_i) \mid v_1 \ldots v_{m_i}.\ \Gamma_i\ \}$ as *calls*. As a convention, we denote calls by upper-case letters C, D, etc. We say that a relation R is *compatible* with a call C iff $C \subseteq R$.

For two calls C and D, we can form their composition $C \circ D$, which intuitively expresses the call C which is immediately followed by D. Interestingly, the composition of two calls can again be written as a call:

$$\begin{aligned}&\{(a\ y,\ b\ y)\ | y.\ P\ y\} \circ \{(c\ x,\ d\ x)\ | x.\ Q\ x\} =\\ &\{(c\ x,\ b\ y)\ | x\ y.\ d\ x = a\ y \wedge Q\ x \wedge P\ y\}\end{aligned}$$

This equation looks wrong at first, but the reader may want to compare this with the definition of relation composition on page 7.

For a termination goal $\mathit{wf}\ (C_1 \cup \ldots \cup C_n :: (\tau \times \tau)\ \mathit{set})$, we call τ the *domain type* of the goal.

Example 3.1. We examine the termination goals arising from a concrete recursive function. The *merge* function below combines two sorted lists to a new sorted list:

$$\begin{aligned}&\mathit{merge}\ \mathit{xs}\ [\,] && = \mathit{xs}\\ &\mathit{merge}\ [\,]\ (y{:}\mathit{ys}) && = y{:}\mathit{ys}\\ &\mathit{merge}\ (x{:}\mathit{xs})\ (y{:}\mathit{ys}) && = \textsf{if}\ x \le y\ \textsf{then}\ x{:}\mathit{merge}\ \mathit{xs}\ (y{:}\mathit{ys})\ \textsf{else}\ y{:}\mathit{merge}\ (x{:}\mathit{xs})\ \mathit{ys}\end{aligned}$$

In open format, the termination goal is as follows:

$$\begin{aligned}&1.\ \mathit{wf}\ ?R\\ &2.\ \bigwedge x\ \mathit{xs}\ y\ \mathit{ys}.\ x \le y \Longrightarrow ((\mathit{xs},\ y{:}\mathit{ys}),\ (x{:}\mathit{xs},\ y{:}\mathit{ys})) \in\ ?R\\ &3.\ \bigwedge x\ \mathit{xs}\ y\ \mathit{ys}.\ \neg\ x \le y \Longrightarrow ((x{:}\mathit{xs},\ \mathit{ys}),\ (x{:}\mathit{xs},\ y{:}\mathit{ys})) \in\ ?R\end{aligned}$$

Converting the goal into the closed format, we obtain

$$\begin{aligned}1.\ \mathit{wf}\ (&\{((\mathit{xs},\ y{:}\mathit{ys}),\ (x{:}\mathit{xs},\ y{:}\mathit{ys}))\ | x\ \mathit{xs}\ y\ \mathit{ys}.\ x \le y\} \cup\\ &\{((x{:}\mathit{xs},\ \mathit{ys}),\ (x{:}\mathit{xs},\ y{:}\mathit{ys}))\ | x\ \mathit{xs}\ y\ \mathit{ys}.\ \neg\ x \le y\})\end{aligned}$$

In the rest of this chapter, we will use the closed format exclusively.

3.1.2 Overview

Our approach for solving these termination goals consists of several components:

Generating measures By means of a configurable set of rules, we collect a set of *measure functions* that map the domain type into the natural numbers. For inductive datatypes, default measure functions are produced by the system. For other types, measures can be declared explicitly.

Proving local descent For the different calls and the generated measures, we try to derive local descent properties automatically. Local descent expresses that a certain measure decreases at a certain call (but it may increase again at another call, hence the attribute *local*). These properties are proved using existing automation in Isabelle. The information about local descent is later used in various ways to search for termination arguments.

Finding termination arguments to simplify the goal Given the local descent proofs, we now search for possibilities to simplify the goal. We look for calls that are strictly decreasing in a certain sense, specified by a so-called *reduction pair*. We may then remove these calls from the goal, provided that the remaining calls are weakly decreasing in the same sense. Reduction pairs are built from the measures, using local descent properties.

From these components, we obtain a termination prover that is already very useful in practice. To this we add two more advanced techniques that further increase the power of the prover:

Using control flow information Another way of simplifying the goal is to use information about the control flow. This information is represented in the form of a control-flow graph. By splitting the graph in its strongly connected components, we can apply a divide-and-conquer strategy to obtain multiple independent subgoals that are simpler than the original one.

Using data flow information Taking into account the data flow in the termination problem, we employ a version of the size-change principle to find stronger termination arguments, also in the form of reduction pairs. This method makes use of a SAT solver to generate the orderings.

We will describe the different steps in a termination proof as a calculus of rules operating on termination goals in closed format. Each rule application produces a (possibly empty) set of subgoals of the same form, and we apply rules until no more subgoals are left — a mode of operation which is familiar to every Isabelle user.

In standard Isabelle terminology, our rules are tactics, since the term *rule* is used for meta-level theorems. However, we prefer to see our proof steps as high-level rules, which also expresses the idea that they are related.

3.2 Related Work

Due to its importance and inherent difficulty, automated termination proving attracts many researchers from different areas. Many approaches have been developed, and doing them all justice is far beyond the scope of this thesis. In the following discussion, we concentrate on approaches that either influenced the method we develop for Isabelle or that have been formalized or applied in a theorem prover.

3.2.1 Termination of term rewrite systems

In the context of term rewriting, termination has been studied intensively in the last decade. Many automated termination provers for term rewrite systems were developed (e.g., [43, 53, 61]) and they compete in a regular termination competition [102].

The formalism of term rewrite systems is attractive for studying termination problems, since it is a very simple model of computation, but, unlike Turing machines, is still relatively abstract, and there is an intuitive correspondence to functional programming.

Notation. We consider untyped first-order terms over a fixed signature. A *term rewrite system* (TRS) $\mathcal{R}$ is a set of pairs (l, r) written as $l \to r$, where l and r are terms. If we view $\mathcal{R}$ as a relation, then its closure under contexts and substitutions is written $\to_{\mathcal{R}}$ and called the *rewrite relation* induced by $\mathcal{R}$. We say that $\mathcal{R}$ *terminates* if $\to_{\mathcal{R}}$ is wellfounded.

The topmost function symbol in a term is calles its *root symbol*. We say that a function symbol is *defined* if it is the root symbol of the left-hand side of some rule in $\mathcal{R}$. Function symbols that are not defined are called *constructors*.

Reduction and simplification orders. One standard way of proving termination of a TRS is to exhibit a *reduction order* $>$, which is a wellfounded order on terms that is closed under contexts and substitution. If it can be shown that $\mathcal{R} \subseteq >$, then also $\to_{\mathcal{R}} \subseteq >$, and hence $\mathcal{R}$ terminates. This gives a natural termination criterion: Given a reduction order, we must only check that each rule in $\mathcal{R}$ is compatible with it.

A *simplification order* is a reduction order that has the *subterm property*, i.e., $\rhd \subseteq >$ for the proper subterm relation $\lhd$. For simplification orders, wellfoundedness directly follows from the subterm property and the closure under context and substitution. Many popular orders like the lexicographic path order and the Knuth–Bendix order fall into this class [7]. However, although simplification orders are popular for termination proofs, many interesting TRSs are not compatible with a simplification order. These TRSs are said to be not *simply terminating*.

Dependency pairs Dependency pairs, introduced by Arts and Giesl [4], are a technique to handle TRSs that are not simply terminating. Instead of directly embedding $\mathcal{R}$ in a simplification order, a set of *dependency pairs* is first generated, which can then be analyzed further using techniques that would fail when used on $\mathcal{R}$ directly.

The set of dependency pairs (DPs) of $\mathcal{R}$ is defined as

$$\begin{aligned} DP(\mathcal{R}) = \{ l^\sharp \to t^\sharp \mid\ & l \to r \in \mathcal{R}, \\ & t \text{ is a subterm of } r \text{ with a defined root symbol } \} \end{aligned}$$

Here, $t^\sharp$ is just the term t whose root symbol is marked with a $^\sharp$, to distinguish it from a normal function symbol. Dependency pairs can again be viewed as a rewrite system over an extened signature that adds a marked version $f^\sharp$ for every defined function symbol f.

When $\mathcal{R}$ is interpreted naively as a functional program, dependency pairs model the recursive calls. The central idea is that it suffices to make sure that there is no infinite sequence of such calls.

A (finite or infinite) sequence of dependency pairs $s_1 \to t_1, s_2 \to t_2, \ldots$ is called a *chain* if there is a substitution σ, such that $\sigma(t_i) \to_{\mathcal{R}}^* \sigma(s_{i+1})$ for all i.

Chains model sequences of recursive calls as they can occur during a reduction. The dependency pairs correspond to recursive calls at the top-level. Between them, there may be further $\to_{\mathcal{R}}$-reductions leading to the next top-level call. Since the dependency pair symbols $f^\sharp$ are distinct from normal function symbols, the intermediate $\to_{\mathcal{R}}$-reductions always occur below the root symbol. A TRS terminates if and only if there are no infinite chains.

Although the property of having no infinite chains is just as undecidable as termnation itself, it is often easier to prove automatically using standard simplification orders or other common techniques. The reason is that it now suffices when the rules of $\mathcal{R}$ are compatible with $\geq$ instead of $>$, and only the dependency pairs must be compatible with $>$. Here, $\geq$ must be a preorder that is closed under contexts and substitutions, $>$ must be wellfounded and closed under substitutions, and $\geq \circ > \circ \geq\ \subseteq\ >$. Such a pair of relations $(>, \geq)$ is called a *reduction pair.*

Operationally, if we have a reduction pair, such that certain DPs are compatible with $>$, and the remaining DPs and the rules of $\mathcal{R}$ are compatible with $\geq$, then the strictly decreasing DPs can be removed. The result is a smaller problem with can be attacked using other reduction pairs or entirely different methods.

Dependency graphs The *dependency graph* of a TRS $\mathcal{R}$ is the graph with the nodes $DP(\mathcal{R})$ and an edge between two DPs, if they form a chain (of length two). The dependency graph serves as a simple model of control flow and allows the decomposition of a termination problem into smaller parts: if it has more than one strongly connected component (SCC), these can be treated separately.

Since it is undecidable in general whether two DPs form a chain, the dependency graph is usually approximated using some safe heuristic. The most commonly used heuristic simply checks if the right hand side of one DP and the left hand side of the other DP are unifiable after replacing all defined symbols with fresh variables and renaming the variables apart.

Dependency pair proofs do not produce a simple characterization of a wellfounded order compatible with all rewrite rules. Instead, they have a goal-oriented structure, where the problems are successively reduced to simpler ones, until they have disappeared.

Example 3.2. Consider the following nonsensical TRS that we want to prove terminating using dependency pairs:

$$\begin{aligned} f(n,0) &\to 0 \\ f(0,s(m)) &\to f(s(f(s(m),s(m))),s(m)) \\ f(s(n),s(m)) &\to s(f(n,m)) \end{aligned}$$

The set of dependency pairs is

$$\begin{aligned} &(1) & f^\sharp(0,s(m)) &\to f^\sharp(s(m),s(m)) \\ &(2) & f^\sharp(0,s(m)) &\to f^\sharp(s(f(s(m),s(m))),s(m))) \\ &(3) & f^\sharp(s(n),s(m)) &\to f^\sharp(n,m) \end{aligned}$$

and the dependency graph looks like this:

$$\begin{tikzcd} (1) \arrow[r, bend left] & (3) \arrow[l, bend left] \arrow[loop above] \arrow[r, bend left] & (2) \arrow[l, bend left] \end{tikzcd}$$

There is only one strongly connected component, so we cannot split the problem into pieces at this point. Now, we note that the second argument is decreasing (3) and stays the same in the other dependency pairs. In rewriting, this is formalized by a subterm ordering and a so-called *argument filtering* that selects the second argument. The rules of the TRS itself are also weakly decreasing in this order, so we may remove (3) from the dependency pairs. Now two dependency pairs remain, but the dependency graph no longer has any nontrivial SCCs:

$$\begin{tikzcd} (1) & & (2) \end{tikzcd}$$

Since there is no edge here, there cannot possibly be a loop, and we are done. If we are used to the idea that we must always find decreasing orders, this step is remarkable, since no order is needed explicitly.

Comparison with Isabelle There is an obvious similarity between the notion of dependency pairs and the calls that form our termination goals: They both correspond to recursive calls in an obvious, albeit informal, functional programming interpretation.

However, some notable differences are hidden under the syntactic similarities:

Deep vs. shallow Most importantly, TRS termination proofs produce well-founded relations on *terms*. The term rewrite system itself, the dependency pairs, and notion of reduction orders are all relations on first-order terms over some fixed signature.

In contrast, the calls in our termination goals are relations on *values* of a certain HOL type, which is the domain type of the function we are defining. These values may be natural numbers, lists or anything else, and although they are represented by Isabelle terms in the system, there is no notion of term inside the logic. Hence, even the basic definitions underlying the methods from rewriting are not meaningful anymore. For example, we cannot express what it means for a relation on natural numbers to be closed under substitution.

This difference is essentially that of syntax vs. semantics and deep vs. shallow embedding, and the difference is not only technical: Giesl [40] shows that naively using term orderings for functional programs is even unsound.

Background theory Isabelle termination goals are interpreted relative to a background theory, which gives meaning to the constants involved in the goal.

A DP problem is interpreted relative to a term rewrite system $\mathcal{R}$, which serves the same purpose, but is syntactically restricted to rewrite rules. Hence, a TRS termination prover can analyze $\mathcal{R}$, and use this information in the proof. In contrast, an Isabelle theory may contain arbitrarily complex definitions, using a variety of specification mechanisms, such as simple equations, inductive definitions, choice operators, or just an axiomatization. As a consequence, the possibilities for analyzing the background theory are very limited.

Equality The notion of a chain in the dependency pairs approach allows arbitrary rewrite steps below the root symbol between two consecutive DP-steps which happen at the root symbol. In the Isabelle termination goals, this role is taken by the equality in the background theory.

The differences in the semantic foundations of both frameworks may appear as just a philosophical concern. But they directly influence the architecture of termination proofs: In our shallow setting, term orders like the lexicographic path order (LPO) cannot be used in the way they are used in rewriting, since they cannot be expressed inside the logic, unless restricted to a fixed type.

Formalizations of term rewriting Two major formalizations of term rewriting have been developed in Coq, both with the aim of certifying termination proofs for term rewrite systems.

The first is the CoLoR project [18] (**Co**q **L**ibrary **o**n **R**ewriting), which is a large formalization of various concepts from rewriting with a focus on termination. It formalizes different term languages including first-order variadic terms, first-order terms over signatures, strings, and simply-typed lambda terms.

The library contains formal versions of the results underlying the dependency pair method. To support graph decomposition, algorithms that compute strongly connected components have been formalized. As base orders (reduction pairs), the library supports polynomial interpretations, different forms of matrix interpretations and the recursive path order.

This setup enables proof reconstruction from certificates produced by an automated termination prover: The automated prover produces certificates following a simple grammar, which are then interpreted by a tool called Rainbow, which produces Coq source files. Using the library, the proofs can then be checked in a fully automated way.

The A3PAT project on the integration of interactive and automated provers goes a similar way with the COCCINELLE library [31]. However, recognizing the overhead of formalizing the graph decomposition algorithms, the authors choose

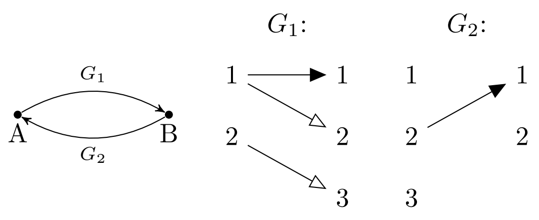


Figure 3.1: A simple size-change problem

a different approach: Instead of working with a deep embedding of rewrite systems, dependency pairs, and graphs, they are represented as inductive relations. This approach avoids formalizing the graph algorithms, since the computations are done on the tactic level [33]. Similar to CoLoR, a tool is provided to convert proof traces from an automated prover into Coq scripts.

In comparison, the approach presented in this chapter is "even more shallow" that that of Coccinelle, since not only graphs and relations use shallow embeddings, but also the *terms* are not a dedicated data type, but general Isabelle/HOL expressions. Due to the inherent gap between shallow and deep embedding, proving termination of recursive functions defined in the logic itself is currently not possible in either of the rewriting-based approaches.

3.2.2 Size-Change Termination

"A program is size-change terminating iff every infinite execution of the program would cause an infinite descent in some well-founded data value." Although its first presentation by Lee, Jones and Ben-Amram [66] was in the context of a simple functional language, this criterion, called *size-change termination* (SCT), is independent from the actual language or programming paradigm used.

We emphasize this generality, which leads to a neat abstraction boundary in our design, by using slightly more general terminology than the original paper.

SCT abstracts from the actual program by viewing it as a set of *control points* and transitions between them, forming a directed graph (the *control graph*). Each control point has a finite set of abstract *data positions* associated with it, which can be seen as slots where runtime data is passed around.

Each transition is labeled with a *size-change graph*, which carries information about the sizes of data. Size-change graphs are usually drawn as bipartite graphs, whose nodes are the data positions of the transition's origin and destination control point. There are two kinds of edges: A *strict* edge, written $p \longrightarrow q$, expresses that the value at data position q (after the transition) is always smaller than the value at position p (before the transition). A *non-strict* or *weak* edge, written $p \rightarrowtriangle q$, means that it is smaller or equal. The absence of an edge means that the value may become larger or the relative sizes are unknown. Figure 3.1 shows a simple size-change instance consisting of two control points A and B, with two and three data positions, respectively.

By connecting the size-change graphs along a control flow path, the data flow becomes visible. Chains of such connected edges are called *threads*. A thread has infinite descent iff it contains infinitely many strict edges.

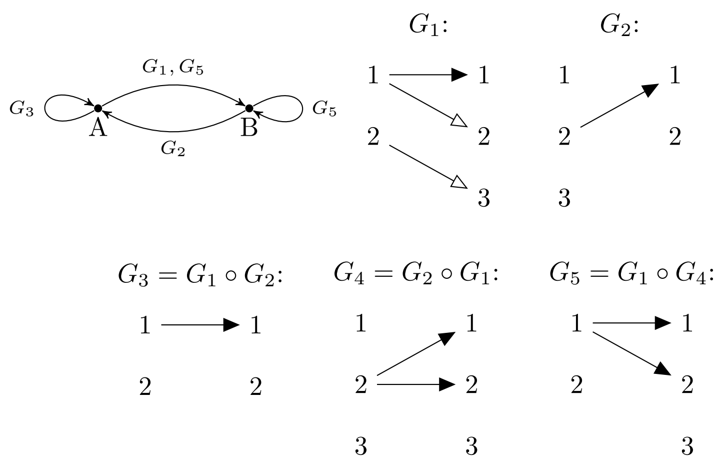


Figure 3.2: Transitive closure of the simple size-change problem

Definition 3.3. *A control graph $\mathcal{C}$ satisfies SCT iff every infinite path has a thread with infinite descent.*

The example in Fig. 3.1 satisfies SCT, since the only infinite path is $A, B, A, B, \ldots$ and it has a thread going through data positions $1, 2, 1, 2, \ldots$, which has infinite descent.

SCT is decidable:

Theorem 3.4. *A control graph $\mathcal{C}$ satisfies SCT iff for every edge in $\mathcal{C}^+$ of the form $p \xrightarrow{G} p$ with $G = G \circ G$, G has an edge of the form $i \longrightarrow i$.*

Here, $\mathcal{C}^+$ is the transitive closure of $\mathcal{C}$. The theorem above suggests an algorithm which simply computes the transitive closure and checks the required property.

Figure 3.2 shows the transitive closure of the size-change problem from Fig. 3.1, where it is easy to check that the relevant graphs (G_3, G_4) have a strict self-edge. We omit details such as the definition of graph composition for the moment and refer to our formalization in §3.10.

Since SCT is a purely combinatorial graph problem, generating size-change problems from programs is a separate issue. Here lies the power of the abstraction: Since nothing is said about what the control points and data positions actually are, the criterion applies to different kinds of programs. The original paper treated simple functional programs, and used functions as control points. Function calls were the transitions, and the data positions were given by the sizes of the function arguments. For imperative programs, one can take program instructions as control points and program variables as data positions as it is done by Avery [6].

Other interpretations are equally valid, as long as (a) infinite executions of the program correspond to infinite paths in the control graph, and (b) the information in the size-change graphs reflects actual size changes in some well-founded data domain. Then a non-terminating execution implies an infinitely decreasing sequence of data values, which is impossible.

SCT has already been applied to term rewriting as well and combined with dependency pairs [103, 104].

Since the size-changes graphs encode knowledge about the data flow in the program, a suitable analysis is required to derive this information. Lee, Jones, and Ben-Amram had a syntactic size analysis in mind, but in fact we have the choice of weapons here, and we will use theorem proving, which does very well on this task.

Together with the relatively compact and self-contained theory, it is the generality and abstraction of the size-change principle, which makes it particularly suited for integration with theorem provers.

Some excitement was generated by a recent approach to termination proofs of low-level systems code like device drivers, implemented in the TERMINATOR tool [32]. The approach of TERMINATOR is based on the fact that a union of wellfounded relations is wellfounded, provided that the whole relation is also transitive — a consequence of Ramseys theorem. This is exploited by generating a whole set of orderings and then using proving that in every execution path from a program point p to itself there is a decrease in at least one of the orderings. The reader might notice the similarity to size-change termination (cf. Thm. 3.4). However, the size-change principle works on abstracted programs in the form of size-change problems where everything is finite, and the transitive closure can therefore be constructed explicitly. TERMINATOR does not use this abstraction and employs model checking techniques to show that $R^+ \subseteq S_1 \cup \ldots \cup S_n$, where R is the transition relation of the program, and S_i are wellfounded orders. Although the underlying abstract theory was formalized in Isabelle by Meng et al. [73] and in HOL4 by Hurd [58], the overall approach could not yet be applied in a theorem proving context, since there is no obvious equivalent to the model checking techniques.

3.2.3 Termination proofs in interactive theorem provers

Compared to the area of term rewriting, termination proof automation in interactive theorem provers is much less developed:

In PVS [85] and Coq [15], no automation exists, and users must supply termination orderings manually. HOL4[1] [44] and HOL Light [51] provide some automation by enumerating all possible lexicographic orderings. For functions with more than five or six arguments, this quickly becomes infeasible. ACL2 [60] uses heuristics to choose a size measure of a single parameter. Lexicographic combinations must be given manually, and are expressed in terms of ordinal arithmetic.

Recently, a more powerful termination criterion has been proposed for ACL2 [69], based on a combination of the size-change principle [66] and other analyses. However, the analysis is nontrivial and only available as an axiomatic extension that must be trusted.

[1]The guessing of termination orderings in HOL4 is unpublished work by Slind

3.3 Measure Functions

Since we center termination proofs around the notion of "becoming smaller", we must develop a suitable notion of size. We will always use natural numbers and their natural order $<$ to measure the size of data items. We use measure functions $m :: \tau \Rightarrow nat$ to assign a size to values of type τ. A measure function m gives rise to a wellfounded relation $(measure\ m) :: (\tau \times \tau)\ set$.

The special role of the natural numbers is an arbitrary design choice, and any type with a wellfounded relation could be used for measuring size. Although it would be more general to use ordinal numbers, which can express more forms of descent directly, it is more practical to use natural numbers:

- To be able to prove any descents for conrete termination problems, we would need to automate at least parts of ordinal arithmetic. Isabelle/HOL currently has no support for this. In contrast, natural numbers are well-supported already, which makes our local descent proofs work out of the box.
- The lower expressive power of measure functions into *nat* is compensated when we combine them to more powerful relations.

3.3.1 Collecting Measure Functions

We provide an interface for the declaration of formation rules that specify how to build measure functions for use in a termination proof. For simplicity, the measure synthesis process is guided only by the domain type of the termination problem. This is only a very crude analysis, but since the generated measure functions are only candidates for use in a proof, it is a useful choice in practice.

To express formation rules for measure functions, we define a predicate on functions:

$$\textit{is-measure} :: (\alpha \Rightarrow nat) \Rightarrow bool$$

This predicate is defined inductively with the sole introduction rule *is-measure* f, such that it trivially holds for any function. It is logically meaningless, but it can be used to express the formation rules for measure functions in HOL itself. For example, we can specify that the function $(nat\ o\ abs) :: int \Rightarrow nat$, which takes the absolute value and coerces it to type *nat*, should be used as a measure on integers:

$$\textit{is-measure}\ (nat \circ abs)$$

This rule is just a normal lemma marked with the attribute [*measure-function*]. Its proof is trivial.

The rules can also be conditional, allowing more flexible specifications. The following two rules specify that in order to measure a pair, we can measure either the left or the right component:

$$\textit{is-measure}\ f \Longrightarrow \textit{is-measure}\ (\lambda p.\ f\ (fst\ p))$$
$$\textit{is-measure}\ f \Longrightarrow \textit{is-measure}\ (\lambda p.\ f\ (snd\ p))$$

In order to generate measure functions for a given type τ, the termination prover can use the standard resolution infrastructure, by setting up a schematic goal

$$\textit{is-measure}\ (?f :: \tau \Rightarrow \textit{nat})$$

and applying the formation rules backwards in Prolog style, which successively instantiates the schematic variable *?f*. Unlike normally, where we are satisfied with the first proof we find, we enumerate all proofs, which corresponds to an enumeration of measure functions.

While the rule-based measure generation is very flexible, it is easily broken by supplying bad rules. For example, if we were to give

$$\textit{is-measure}\ f \Longrightarrow \textit{is-measure}\ f$$

we would make the derivations loop immediately. It is the user's responsibility to declare only well-behaved rules. In particular, the set of solutions must be finite, and no solution may have any schematic variables.

Adding more measure functions can only increase the power of the termination prover, but we pay for this with a larger search space. Some rules are particularly expensive, as they can lead to an exponential blowup in the number of measure functions. For example, we could measure products by the sum of the measures of their components:

$$\textit{is-measure}\ f \Longrightarrow \textit{is-measure}\ g \Longrightarrow \textit{is-measure}\ (\lambda(x,\ y).\ f\,x + g\ y)$$

Now if we have a large product $\alpha_1 \times \ldots \times \alpha_n$, and each type α_i generates two measure functions, this rule generates a total of 2^n different combinations, which may be prohibitive.

3.3.2 Size functions for inductive datatypes

For each inductive datatype definition, Isabelle automatically defines a corresponding size function using primitive recursion. Below is the size function for lists:

$$\begin{aligned}\textit{list-size}\ [\,] &= \mathit{0}\\ \textit{list-size}\ (x{:}xs) &= \textit{list-size}\ xs + \textit{Suc}\ \mathit{0}\end{aligned}$$

Note that the non-recursive constructor [] has size zero. This is an arbitrary choice motivated by the historical development. It has the advantage that the size of a natural number is the number itself and the size of a list is its length.

The definition of *list-size* above is "shallow" in the sense that it only counts the constructors in the spine of the list, but ignores the actual elements. This is often just what we want, as it captures structural recursion on lists.

However, a technical issue arises in the presence of nested datatypes. Consider the datatype of n-ary trees:

$$\textbf{datatype}\ \alpha\ \textit{tree} = \textit{Node}\ \alpha\ (\alpha\ \textit{tree list})$$

How should we define the size of such a tree? Since the *Node* constructor takes a list, it is tempting to simply define

$$\textit{tree-size}\ (\textit{Node}\ v\ xs) = \textit{list-size}\ xs + \textit{Suc}\ \mathit{0}$$

This definition does not capture the recursive structure of trees, since *list-size* ignores the recursive occurrences of subtrees, which we should be counted as well.

So the size of an α *tree list* must apparently be measured differently than that of an α *list*. Before Isabelle 2007, this was done by producing a new function *tree-list-size* with the expected behaviour. The functions *tree-size* and *tree-list-size* are then mutually recursive:

$$\begin{aligned}
&\textit{tree-list-size}\ [\,] &&= \mathit{0}\\
&\textit{tree-list-size}\ (x{:}xs) &&= \textit{tree-size}\ x + \textit{tree-list-size}\ xs + \mathit{Suc}\ \mathit{0}\\
&\textit{tree-size}\ (\mathit{Node}\ v\ xs) &&= \textit{tree-list-size}\ xs + \mathit{Suc}\ \mathit{0}
\end{aligned}$$

From these definition we get a useful size measure. The problem is that we also need a lemma that connects the two functions:

$$x \in \mathit{set}\ xs \Longrightarrow \textit{tree-size}\ x < \textit{tree-list-size}\ xs$$

This lemma requires induction and had to be stated and proved manually before any termination proofs over trees would work — an unpleasant situation.

A better solution is to add an extra function parameter to *list-size* which is used to measure the elements of the list:

$$\textit{gen-list-size} :: (\alpha \Rightarrow \mathit{nat}) \Rightarrow \alpha\ \mathit{list} \Rightarrow \mathit{nat}$$

$$\begin{aligned}
&\textit{gen-list-size}\ f\ [\,] &&= \mathit{0}\\
&\textit{gen-list-size}\ f\ (x{:}xs) &&= f\ x + \textit{gen-list-size}\ f\ xs + \mathit{Suc}\ \mathit{0}
\end{aligned}$$

We can then define the simpler function *list-size* as *gen-list-size* $(\lambda x.\ \mathit{0})$. The advantage is that we can also use *gen-list-size* in the definition of *tree-size*:

$$\textit{tree-size}\ (\mathit{Node}\ v\ xs) = \textit{gen-list-size}\ \textit{tree-size}\ xs + \mathit{Suc}\ \mathit{0}$$

This definition is equivalent to the previous one, but it no longer requires mutual recursion and the somewhat ad-hoc *tree-list-size*. As a practical benefit, the lemma above now becomes generic and no longer mentions trees:

$$x \in \mathit{set}\ xs \Longrightarrow f\ x < \textit{gen-list-size}\ f\ xs$$

This generic lemma applies to all datatypes that use recursion under the type constructor *list*. It is part of the standard library, together with similar lemmas about products and options.

The generated size functions are automatically declared as measures using the mechanism described above. Not all of these measures are always useful though: For non-recursive types like the product type, the size function is the constant zero function. A better measure function can of course still be declared manually.

Generalized size functions of the type *gen-list-size* are not declared as measure functions automatically, since they can lead to an exponential search space if the type has arity greater than one. For the the common case of lists, we declare the rule manually:

$$\textit{is-measure}\ f \Longrightarrow \textit{is-measure}\ (\textit{gen-list-size}\ f)$$

Example 3.5. For domain type $\tau = \mathit{int} \times \mathit{nat\ list}$, the formation rules described above produce the measure functions $\lambda p.\ (\mathit{nat} \circ \mathit{abs})\ (\mathit{fst}\ p)$, $\lambda p.\ \textit{list-size}\ (\mathit{snd}\ p)$, and $\lambda p.\ \textit{gen-list-size}\ \textit{nat-size}\ (\mathit{snd}\ p)$. Note that *nat-size* is just the identity.

3.4 Proving Local Descent

The type-based measure synthesis produces candidates for measure functions. To determine whether a measure is actually helpful with a given termination problem, we have to see whether it makes some calls decreasing.

For a call $C = \{(r,\, l) \mid v_1 \ldots v_m.\ \Gamma\}$, the *strict descent property* with respect to a pair of measures (m_1, m_2) is the formula

$$\bigwedge v_1 \ldots v_m.\ \Gamma \Longrightarrow m_2\ r < m_1\ l\,,$$

and the *weak descent property* is

$$\bigwedge v_1 \ldots v_m.\ \Gamma \Longrightarrow m_2\ r \leq m_1\ l\,.$$

Unsurprisingly, the descent properties are undecidable, and we cannot expect a general checker for them.

If we were developing a stand-alone termination prover, we would now face the problem of finding a syntactic criterion about r, l, and Γ (e.g., a structural subterm property) that implies descent for some fixed measures. If the check is positive, we must produce a proof of the descent property.

Now, since we live in the Isabelle framework, which provides ample proof automation, we can use the available generic tools for this task. For checking descent, we simply state the strict descent property and invoke the *auto* method. If the proof succeeds, we store the theorem for later use. If it fails, we try the same for the weak descent property. If this also fails, we conservatively assume that the conditions do not hold.

This strategy of trial-and-error proving is somewhat unusual, and we have not seen it in other contexts. Relying on the available infrastructure has two notable advantages:

First, the *auto* method, which combines simplification with classical reasoning and arithmetic, has been subject to extensive tuning, since it is one of the most widely-used methods in the system. Thus it is likely to be more effective than a specialized prover, since the form of decrease conditions is relatively general.

Second, since the method can be configured with additional rule declarations, we obtain a simple way of supplying lemmas to the descent prover. Descents that are not provable on their own may become provable when using an existing lemma. This is a crucial feature, since the descent conditions may refer to arbitrary user-defined concepts.

Due to the specific structure of the descent condition, it helps to add some rules to the simplification set of the automated prover. Many of these rules are normally not added since they increase the search space and may slow down simplification. However, since local descent proofs are typically not very large, we can pay this price for the improved automation. Figure 3.3 lists these rules.

Result matrices For each call C, the results of the proofs are stored in a matrix M^C, with $M^C_{i,j} \in \{<, \leq, ?\}$, indexed by the measure functions. The entries $<$ and $\leq$ stand for a successful strict or weak descent proof, and ? denotes a failure of both proofs.

Example 3.6. Recall the *merge* function from §3.1.1, and let us suppose that the measure generation process has produced the functions $m_1 = (\lambda p.\ \textit{list-size}$

$(\bigwedge x.\ x \in \mathit{set\ xs} \Longrightarrow f\ x < g\ x) \Longrightarrow \mathit{gen\text{-}list\text{-}size}\ f\ \mathit{xs} \leq \mathit{gen\text{-}list\text{-}size}\ g\ \mathit{xs}$
$x \in \mathit{set\ xs} \Longrightarrow y \leq f\ x \Longrightarrow y \leq \mathit{gen\text{-}list\text{-}size}\ f\ \mathit{xs}$
$x \in \mathit{set\ xs} \Longrightarrow y < f\ x \Longrightarrow y < \mathit{gen\text{-}list\text{-}size}\ f\ \mathit{xs}$
$\mathit{prod\text{-}size}\ f\ g\ p = f\ (\mathit{fst}\ p) + g\ (\mathit{snd}\ p) + \mathit{Suc}\ 0$
$m \leq n \Longrightarrow m < \mathit{Suc}\ n$
$x < y \Longrightarrow x \leq y$
$x \leq z \Longrightarrow x \leq y + z$
$x \leq y \Longrightarrow x \leq y + z$
$x < z \Longrightarrow x < y + z$
$x < y \Longrightarrow x < y + z$

Figure 3.3: Extra simplification rules for local descent proofs

$(\mathit{fst}\ p))$ and $m_2 = (\lambda p.\ \mathit{list\text{-}size}\ (\mathit{snd}\ p))$. The strict descent property for the first call and (m_1, m_1) is

$$\bigwedge x\ \mathit{xs}\ y\ \mathit{ys}.\ x \leq y \Longrightarrow \mathit{list\text{-}size}\ (\mathit{fst}\ (\mathit{xs},\ y{:}\mathit{ys})) < \mathit{list\text{-}size}\ (\mathit{fst}\ (x{:}\mathit{xs},\ y{:}\mathit{ys}))\,.$$

This is easily proved automatically by simplifying with $\mathit{fst}\ (a,\ b) = a$ and then unfolding *list-size* on the right-hand side, which results in a statement that is trivial for the built-in arithmetic:

$$\bigwedge x\ \mathit{xs}\ y\ \mathit{ys}.\ x \leq y \Longrightarrow \mathit{list\text{-}size}\ \mathit{xs} < \mathit{list\text{-}size}\ \mathit{xs} + \mathit{Suc}\ 0$$

If we take the measure combination $(m_2,\ m_2)$, strict descent is not provable, but weak descent is. For the combination $(m_1,\ m_2)$, none of the conditions is provable. Here is an attempt to prove weak descent:

$$\bigwedge x\ \mathit{xs}\ y\ \mathit{ys}.\ x \leq y \Longrightarrow \mathit{list\text{-}size}\ (\mathit{fst}\ (\mathit{xs},\ y{:}\mathit{ys})) \leq \mathit{list\text{-}size}\ (\mathit{snd}\ (x{:}\mathit{xs},\ y{:}\mathit{ys}))$$

This time, simplification yields

$$\bigwedge x\ \mathit{xs}\ y\ \mathit{ys}.\ x \leq y \Longrightarrow \mathit{list\text{-}size}\ \mathit{xs} \leq \mathit{list\text{-}size}\ \mathit{ys} + \mathit{Suc}\ 0\ ,$$

where we are stuck and must abandon the proof attempt.

Collecting the results of all proofs yields the following local descent matrices:

$$M^{C_1} = \begin{pmatrix} < & ? \\ ? & \leq \end{pmatrix} \qquad M^{C_2} = \begin{pmatrix} \leq & ? \\ ? & < \end{pmatrix}$$

3.5 Simple Termination Proofs and Lexicographic Descent

Up to now, we were just collecting the necessary information to find a termination proof, but we have not yet touched the goal. In this section, we introduce two simple rules to build termination proofs from the information we have.

The first rule is used to conclude the proof when we are done and no more calls are left:

Rule 3.7 (Trivial goal)**.** *The goal wf* $\emptyset$ *is trivial and can be discharged immediately.*

The second rule is more interesting:

Rule 3.8 (Reduction Pair)**.** *Given a reduction pair $(R,\ S)$, we may remove calls from the goal that are compatible with R, provided that the remaining calls are compatible with S.*

The formal justification for this step is the following straightforward consequence of the union lemma (Lemma 1.1):

$$\frac{\textit{reduction-pair}\ (R,\ S) \qquad C \subseteq R \qquad D \subseteq S \qquad \textit{wf}\ D}{\textit{wf}\ (C \cup D)}$$

To apply the rule, we must find a useful reduction pair, which is the creative part of termination proving. For now, we will use only reduction pairs that arise from single measure functions. In §3.8, more sophisticated reduction pairs will be used.

In the results of the local descent proofs, we look for a measure m, such that $M^{C}_{m,m} = <$ for at least one call C, and $M^{C'}_{m,m} \in \{<, \leq\}$ for all the other calls. Then we use *measure-rp m* as a reduction pair. The premises $C \subseteq R$ and $D \subseteq S$ of the lemma above follow directly from the local descent properties that we have already proved.

Example 3.9. We continue with the *merge* example, where the goal is still in its initial form:

$$\begin{array}{l} \textit{1. wf}\ (\{((xs,\ y{:}ys),\ (x{:}xs,\ y{:}ys))\ |x\ xs\ y\ ys.\ x \leq y\} \cup \\ \qquad \{((x{:}xs,\ ys),\ (x{:}xs,\ y{:}ys))\ |x\ xs\ y\ ys.\ \neg\ x \leq y\}) \end{array}$$

From the descent matrices we can see that we can use the measure m_1 to remove C_1, since $M^{C_1}_{1,1} = <$ and $M^{C_2}_{1,1} = \leq$. The premises of the reduction pair lemma are easily reduced to the local descent properties, e.g.,

$$\{((xs,\ y{:}ys),\ x{:}xs,\ y{:}ys)\ |x\ xs\ y\ ys.\ x < y\} \subseteq \textit{fst}\ (\textit{measure-rp}\ (\lambda p.\ \textit{list-size}\ (\textit{fst}\ p)))$$

$$\rightsquigarrow \bigwedge x\ xs\ y\ ys.\ x < y \Longrightarrow ((xs,\ y{:}ys),\ (x{:}xs,\ y{:}ys)) \in \textit{fst}\ (\textit{measure-rp}\ (\lambda p.\ \textit{list-size}\ (\textit{fst}\ p)))$$

$$\rightsquigarrow \bigwedge x\ xs\ y\ ys.\ x < y \Longrightarrow \textit{list-size}\ (\textit{fst}\ (xs,\ y{:}ys)) < \textit{list-size}\ (\textit{fst}\ (x{:}xs,\ y{:}ys))$$

Here we just had to unfold the definition of *measure* and the comprehension to reduce the inclusion to the statement that we have already proved. We discharge this goal using the stored local descent theorem.

Now there is just one call left:

$$\textit{1. wf}\ \{((x{:}xs,\ ys),\ (x{:}xs,\ y{:}ys))\ |x\ xs\ y\ ys.\ \neg\ x \leq y\}$$

We solve this call in the same way as the first one, but using the measure m_2 this time. This reduces the goal to *wf* $\emptyset$, which we then discharge using Rule 3.7.

A note on nested functions The calls above did not contain references to functions other than constructors, this is not always the case, since recursive calls may contain arbitrary user-defined functions (often called *nested* functions, but not in the sense of §2.4). Then, lemmas about these functions are required that express that, e.g., the functions never increase the size of their arguments. A typical example is the inequality

$$\textit{list-size}\ (\textit{filter}\ P\ xs) \leq \textit{list-size}\ xs$$

Such lemmas usually require induction, and we make no attempt to discover and prove them automatically.

However, the local descent proofs will use these lemmas, if they are already present in the theory. Here we are very much in the tradition of automated proof tools in Isabelle, which rely heavily on their configuration and lemmas in the background theory.

Walther [111] describes techniques for discovering and proving such lemmas automatically in a first-order setting, using a specialized estimation calculus. The dependency pairs method deals with nested functions implicity by requiring that reduction pairs must also be compatible with the rules of $\mathcal{R}$ (or rather, a certain subset).

However, both approaches have an implicit closed world assumption: They require that the definitions of the nested functions follow a certain form that can be analyzed. In an open logical framework like Isabelle, this is not true in general. Nevertheless, in many cases, lemmas about nested functions could probably be speculated and proved automatically, which would further improve the methods presented here. However, we do not adress this problem further, and rely on the user to provide the relevant lemmas about nested functions.

The method we have developed up to this point already supports lexicographic descent naturally, since we can apply Rule 3.8 multiple times, each time with a different reduction pair. This approach is implemented in Isabelle as the method *lexicographic-order*[2]. In the following sections, we develop other approaches, which improve upon this basic approach.

3.6 Control Flow: Dependency Graph Analysis

In this section we will analyze the control flow inherent in a termination problem by using a form of abstract control flow graph which, following terminology from term rewriting, is called the *dependency graph*. The dependency graph is a graph between calls. Two calls C and D are connected with an edge, if C can be immediately followed by D.

3.6.1 Building the Dependency Graph

To determine whether a call can follow another, we employ a similar trial-and-error strategy as we did for local descent in §3.4. For each combination of two calls, we set up the goal

$$C \circ D = \emptyset$$

which expresses that a transition in C can never be followed by a transition in D. If we can prove the goal (again using *auto*), the resulting theorem is

[2]The implementation of *lexicographic-order* is actually different, but equivalent. It works on open goals instead of closed ones, constructing a lexicographic termination order explicitly [26]. It was implemented before the more general framework using closed goals existed.

our justification for not putting the respective edge in the dependency graph. Otherwise we are conservative and draw an edge.

Example 3.10. Consider the artificial function *bar*, which has has two modes: Depending on a boolean flag, either the first or the second argument decreases:

$$\begin{aligned}
&\mathit{bar}\ \mathit{True}\ (\mathit{Suc}\ n)\ m &&= \mathit{bar}\ \mathit{True}\ n\ (\mathit{Suc}\ m)\\
&\mathit{bar}\ \mathit{True}\ 0\ m &&= \mathit{bar}\ \mathit{False}\ 0\ m\\
&\mathit{bar}\ \mathit{False}\ n\ (\mathit{Suc}\ m) &&= \mathit{bar}\ \mathit{False}\ (\mathit{Suc}\ n)\ m\\
&\mathit{bar}\ \mathit{False}\ n\ 0 &&= n
\end{aligned}$$

The corresponding termination problem has three calls ($C_1 \cup C_2 \cup C_3$):

$$\begin{aligned}
&\mathit{wf}\ (\{((\mathit{True},\ n,\ \mathit{Suc}\ m),\ (\mathit{True},\ \mathit{Suc}\ n,\ m))\ |n\ m.\ \mathit{True}\} \cup\\
&\quad \{((\mathit{False},\ 0,\ m),\ (\mathit{True},\ 0,\ m))\ |m.\ \mathit{True}\} \cup\\
&\quad \{((\mathit{False},\ \mathit{Suc}\ n,\ m),\ (\mathit{False},\ n,\ \mathit{Suc}\ m))\ |n\ m.\ \mathit{True}\})
\end{aligned}$$

We obtain the following dependency graph:

$$\begin{tikzcd}
\boxed{\{((\mathit{True},\ n,\ \mathit{Suc}\ m),\ (\mathit{True},\ \mathit{Suc}\ n,\ m))\ |n\ m.\ \mathit{True}\}} \arrow[loop left] \arrow[d] \\
\boxed{\{((\mathit{False},\ 0,\ m),\ (\mathit{True},\ 0,\ m))\ |m.\ \mathit{True}\}} \arrow[d] \\
\boxed{\{((\mathit{False},\ \mathit{Suc}\ n,\ m),\ (\mathit{False},\ n,\ \mathit{Suc}\ m))\ |n\ m.\ \mathit{True}\}} \arrow[loop left]
\end{tikzcd}$$

For each edge that is absent from the graph, we have proved a property of the form $C \circ D = \emptyset$ as justification. Again, we rely on available automation for the proofs. For example, here is the proof of $C_2 \circ C_1 = \emptyset$:

$$\begin{aligned}
&\{((\mathit{False},\ 0,\ m),\ (\mathit{True},\ 0,\ m))\ |m.\ \mathit{True}\} \circ\\
&\{((\mathit{True},\ n,\ \mathit{Suc}\ m),\ (\mathit{True},\ \mathit{Suc}\ n,\ m))\ |n\ m.\ \mathit{True}\}\\
&= \{((\mathit{True},\ n',\ \mathit{Suc}\ m'),\ (\mathit{True},\ 0,\ m))\ |m\ m'\ n'.\\
&\ (\mathit{True},\ \mathit{Suc}\ n',\ m') = (\mathit{False},\ 0,\ m)\}\\
&= \{((\mathit{True},\ n',\ \mathit{Suc}\ m'),\ (\mathit{True},\ 0,\ m))\ |m\ m'\ n'.\\
&\ \mathit{True} = \mathit{False} \wedge \mathit{Suc}\ n' = 0 \wedge m' = m\}\\
&= \{((\mathit{True},\ n',\ \mathit{Suc}\ m'),\ (\mathit{True},\ 0,\ m))\ |m\ m'\ n'.\ \mathit{False}\}\\
&= \emptyset
\end{aligned}$$

3.6.2 Decomposition

The dependency graph above reflects the control flow of *bar*. There are two separate loops, and for a termination proof, we should be able to treat them independently. We introduce the following rule to do just that.

Rule 3.11 (Decomposition)**.** *If the dependency graph of the goal is not strongly connected, we can split the goal into independent subgoals, corresponding to the strongly connected components (SCCs) of the dependency graph.*

The decomposition rule is formally justified by the union lemma. If we have more than one SCC, we can always partition the calls in to an upper part U and a lower part L, such that no call in the upper part is reachable from the lower part, i.e.,

$L \circ U = \emptyset$.

In particular, this implies $L \circ U \subseteq U$, which is the premise of the union lemma. Using this lemma, we split the termination goal in two independent subgoals. We can repeat this step until we have separated all SCCs.

Example 3.12. Continuing with the example above, we use Rule 3.11 to split the termination problem in three parts. First, we split off $U = C_1$ from $L = C_2 \cup C_3$. We have to show that $(C_2 \cup C_3) \circ C_1 = \emptyset$, which follows from the dependency graph using distributivity. Now we have two independent subgoals. The subgoal *wf* $(C_2 \cup C_3)$ can be treated in the same way, and we end up with the proof where all calls are separated:

1. $wf\ \{((True,\ n,\ Suc\ m),\ (True,\ Suc\ n,\ m))\ |n\ m.\ True\}$
2. $wf\ \{((False,\ 0,\ m),\ (True,\ 0,\ m))\ |m.\ True\}$
3. $wf\ \{((False,\ Suc\ n,\ m),\ (False,\ n,\ Suc\ m))\ |n\ m.\ True\}$

3.6.3 Trivial calls

After splitting the problem into its strongly connected components, we may end up with a call that is not reachable from itself. This corresponds to a dependency graph with a single node and no edge. Intuitively, we do not need to consider such calls, as they cannot lead to nontermination:

Rule 3.13 (Trivial Call)**.** *A goal with a single call that is not reachable from itself can be discarded.*

Formally, we use the self-composition lemma (Lemma 1.3) to reduce the goal *wf* C to *wf* $(C \circ C)$. Since there is no edge, we know that $C \circ C = \emptyset$, which is trivially wellfounded.

In our running example, Rule 3.13 allows us to drop C_2.

3.7 Mutual Recursion

As we have seen in §2.6.4, the function package reduces mutually recursive functions to a single function on a sum type. Thus, the termination goal for mutually recursive functions involves relations over that sum type.

For example, the following goal arises from the definition of *even* and *odd*:

$$1.\ wf\ (\{(Inr\ n,\ Inl\ (Suc\ n))\ |n.\ True\} \cup \\ \{(Inl\ n,\ Inr\ (Suc\ n))\ |n.\ True\})$$

In general, a call from f to g in a mutual recursive definition has the form $\{\ (Inj_g\ r_i,\ Inj_f\ l_i)\ |\ v_1 \ldots v_{m_i}.\ \Gamma_i\ \}$, where Inj_f and Inj_g denote the injections that map the argument of the functions f and g into the sum type. They consist of compositions of the basic injections *Inl* and *Inr*.

The easiest way of supporting mutual recursion is by ignoring it and keeping the analysis unmodified. All we have to do is declare a rule to generate measure functions on sum types. An obvious candidate is the following rule, which combines two measures on α and β, respectively, and yields a measure on $\alpha + \beta$.

$$is\text{-}measure\ f \Longrightarrow is\text{-}measure\ g \Longrightarrow is\text{-}measure\ (sum\text{-}case\ f\ g)$$

For the type *nat* + *nat* this rule produces the measure *sum-case nat-size nat-size*, which is just fine for *even* and *odd*. Intuitively, it generates all measures for each of the individual functions, and then builds all possible combinations. As we mentioned earlier, this approach is very expensive: The number of measures it produces is exponential in the size of the sum type.

Recall that the number of proof attempts that we have to perform to check for local descent is quadratic in the number of measures. Since proof attempts are relatively expensive, the use of the rule above is already problematic for three or four mutually recursive functions with a few arguments each. Thus, trying to avoid the exponential behaviour is clearly a good investment.

3.7.1 Avoidable exponential blowup

Concerning the number of proof attempts, the exponential blowup can be avoided, since trying all measure combinations is highly redundant: In a call from f to g, it is completely irrelevant what measure we assign to all other functions, and sum measures differing only in components other than f and g will not change the outcome of this particular descent proof.

It is a better strategy to treat the outermost sum type structure as special. If the domain type has the form

$$\tau_1 + \tau_2 + \ldots + \tau_n$$

we produce a separate set of measures $\mathcal{M}_i$ for each component τ_i.

For the local descent proofs we now remove the injections from the calls, and directly insert the measures that we have generated for the respective sum component. So for the call $\{\ (\mathit{Inj}_g\ r,\ \mathit{Inj}_f\ l) \mid v_1 \ldots v_m.\ \Gamma\ \}$ we construct the local descent goal

$$\bigwedge v_1 \ldots v_m.\ \Gamma \Longrightarrow m_2\ r < m_1\ l$$

where $m_1 \in \mathcal{M}_f$ and $m_2 \in \mathcal{M}_g$. The goal for weak descent is analogous.

As before, we build a matrix for each recursive call, where we collect the results of local descent proofs. The difference is that the matrices are no longer square matrices but rectangular, since different sets of measures may be used depending on the domain type of the different functions. This approach avoids the exponential blowup, while still collecting exactly the same information.

3.7.2 Unavoidable exponential blowup

Finding a suitable global measure on the sum type that works for all calls is now a purely combinatorial task. Abstracting from functions and measures and the like, we have the following problem:

> Assignment: Given finite sets F and M, and a relation $R \subseteq F \times F \times M \times M$, is there a mapping $f : F \to M$ such that $\forall x, x' \in F.\ (x, x', f(x), f(x')) \in R$?

The abstraction is as follows: F is the set of functions, and M is the set of basic measures, where we assume without loss of generality that the number of basic measures is the same for all functions (e.g., by taking the maximum). The relation R encodes the constraints given by the local descent proofs: $(f, g, i, j) \in R$ iff for some call C from f to g, $M^C_{ij} \in \{<, \leq\}$, (or $M^C_{ij} = <$, if that call should be strictly decreasing).

The Assignment problem has the typical structure of an NP-complete problem, and indeed it is:

Lemma 3.14. *The* Assignment *problem is NP-complete.*

Proof. We give a reduction from the 3-colouring problem for graphs, which asks if the nodes of a given graph can be marked with three colours, such that any two adjacent nodes get different colours. This problem is known to be NP-complete [87].

For a graph $G = (V, E)$, we define the problem instance

$$F = V, \qquad M = \{\text{Red}, \text{Green}, \text{Blue}\}, \qquad R = \{(v, v', c, c') \mid (v, v') \notin E \vee c \neq c'\}.$$

Now every solution f for Assignment is a valid colouring: If $(v, v') \in E$ and $f(v) = c$ we have $(v, v', c, c) \notin R$ and thus $f(v') \neq c$. Conversely, every colouring is clearly a valid Assignment. Thus our problem is NP-hard, and since checking a given solution is trivial, it is also NP-complete. □

The following lemma shows that any instance of Assignment can actually arise from a termination goal, hence our problem is not overly general:

Lemma 3.15. *For any instance of* Assignment*, there is a termination problem that can be simplified using a combination of measures, if and only if the combination problem has a solution.*

Proof. Given an instance (F, M, R) of Assignment, we produce a termination problem with $n = |F|$ functions, each taking $k = |M|$ arguments of type *nat*. Thus the domain type of the termination problem is an n-fold sum of k-fold products. We use the measures $m_1, \ldots, m_k$ where m_i is the projection of a k-tuple on the ith argument.

For each pair $f, g \in F$, we construct the call

$$C_{fg} = \{\ (\mathit{Inj}_g\ (\mathit{min}\ S_1,\ \ldots,\ \mathit{min}\ S_m),\ \mathit{Inj}_f\ (x_1,\ \ldots,\ x_m))\ \mid x_1 \ldots x_m.\ \mathit{True}\ \}$$

where S_i abbreviates the finite set of all x_j such that $(f, g, i, j) \in R$. Furthermore, we pick an arbitrary function $f \in F$ and add the call

$$C_< = \{\ (\mathit{Inj}_f\ (x_1,\ \ldots,\ x_m),\ \mathit{Inj}_f\ (\mathit{Suc}\ x_1,\ \ldots,\ \mathit{Suc}\ x_m))\ \mid x_1 \ldots x_m.\ \mathit{True}\ \}.$$

For any assignment of measures to functions, the call $C_<$ is strictly decreasing. The call C_{fg} is weakly decreasing for exactly those pairs of measures m_i (assigned to f) and m_j (assigned to g) where $(f, g, i, j) \in R$. □

This result shows that mutually recursive functions add a new source of complexity to the termination proving task, since the measures used for the different functions must accomodate the data flow between them, which is an NP-complete problem. Strictly speaking, these issues are not really connected to mutual recursion at all, since the sum types could also appear explicitly in the definition of a single function. However, such a definition would not be very natural.

Fortunately, the times when NP-completeness was a synonym for "abandon all hope..." are gone, especially when the main problem we are working at is undecidable. Clearly, the ASSIGNMENT problem above wants to be encoded into SAT and given to the latest SAT solver, which can handle such combinatorial problems best. In the next section, we will do this, but with a problem that is more general. After that, finding combinations for sum types is an instance of that problem.

3.8 Data Flow: Size-Change Termination with Certificates

The termination prover we have developed so far does not handle functions that permute their arguments in a recursive call. Consider the following function taken from the original paper on size-change termination [66]:

```
perm m n r =
(if 0 < r then perm m (r − 1) n
 else if 0 < n then perm r (n − 1) m else m)
```

Here is the corresponding termination goal:

```
1. wf ({((m, r − 1, n), (m, n, r)) |m n r. 0 < r} ∪
       {((r, n − 1, m), (m, n, r)) |m n r. ¬ 0 < r ∧ 0 < n})
```

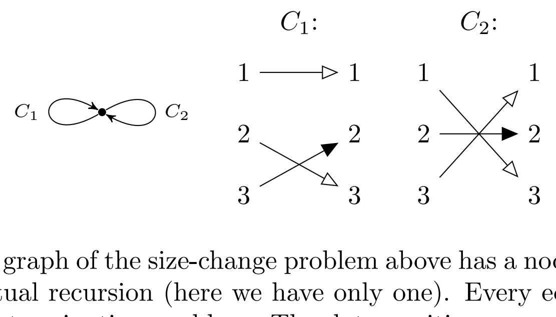
If we use the projections as measures as usual, we can derive the following local descent matrices:

$$M^{C_1} = \begin{pmatrix} \leq & ? & ? \\ ? & ? & < \\ ? & \leq & ? \end{pmatrix} \qquad M^{C_2} = \begin{pmatrix} ? & ? & \leq \\ ? & < & ? \\ \leq & ? & ? \end{pmatrix}$$

Our current way of building reduction pairs only uses the information at the diagonals of the call matrices, but this is not sufficient for this example. What we need is a combination of the individual measures.

For this particular goal, the measure $\lambda(x,\, y,\, z).\ x + y + z$ is obviously sufficient, and declaring an appropriate measure formation rule will help. However, we will now develop a more general method.

Like in the previous section, we are facing a data flow problem, since we need a way to track what happens to the values in the recursive calls. This information, which we have represented as matrices, can also be depicted as size-change graphs, which nicely visualize the data flow:



The control graph of the size-change problem above has a node for each function in the mutual recursion (here we have only one). Every edge corresponds to a call in the termination problem. The data positions associated with each control point represent the measures. The size-change graph for a call contains the same information that we have previously encoded in a matrix.

Note that the control graph in this size-change instance is not the dependency graph we studied before: Here, calls are represented as edges, not as nodes. In a graph-theoretic sense, the dependency graph is dual to the control graph used here (possibly with some edges missing due to the finer approximation). As we shall see later, this is not the only way of building size-change problems from termination goals, but it is the simplest one and it allows for easy proof reconstruction.

The reader may want to check that the size-change problem above actually satisfies SCT. Using Thm. 3.4, this involves computing the transitive closure which produces quite a number of graphs, corresponding to the different ways of permuting the arguments. The property is easier to see using Def. 3.3 directly: In this problem instance, any infinite path has exactly three infinite threads that are never interrupted. At each position, exactly one of the threads has a strict edge. Thus, there must be one thread that has infinitely many strict edges.

3.8.1 Certificates for size-change termination?

Things would be simplest if we could generate short certificates of some kind, which prove that a function is size-change terminating, and which can be easily checked. This is the case for the reduction pairs we used so far: Using the local descent properties it is easy to check that the calls are decreasing. One would wish that we could construct reduction pairs for size-change termination in a similar way.

However, a complexity argument shows that such certificates are unlikely to exist, due to the PSPACE-hardness result for SCT [66]:

Corollary 3.16. *If there were certificates proving $x \in SCT$ that could be checked in polynomial time, then PSPACE = NP, which complexity theorists find unlikely [87].*

Proof. Assume that such certificates exist, then SCT $\in$ NP by a simple guess-and-check argument. But SCT is PSPACE-hard, thus PSPACE $\subseteq$ NP. □

If we assume that proof checking is polynomial in the size of the proof, this means that any certificates must be of exponential size. Of course this does not exclude that there could be certificates that are long but still conceptually simple.

Lee [65] gives an explicit construction of a global measure function for a size-change problem. However, the construction is triple-exponential in size. Recently, this could be reduced to single-expontential complexity [12], which is still not very attractive as a certificate scheme.

3.8.2 SCNP = SCT in NP

Instead of using full size-change termination, we restrict ourselves to a subset of SCT that lies in NP and is consequently called SCNP. The restriction was proposed by Ben-Amram and Codish [11], and they show how to use a SAT-solver to produce certificates in the form of mappings into certain wellfounded orders. A formal termination proof must then only check that each call is decreasing. By definition, the class SCNP contains the SCT instances that admit such certificates. Since this definition is a little unnatural, it is perhaps more appropriate to view SCNP as an incomplete decision procedure for SCT, instead of an independent problem class. Experimenting on two test suites, Ben-Amram and Codish show that SCNP is quite powerful: It solves all size-change problems in the test suites, except for those intentionally constructed as counterexamples.

Lifted orders: max, min, and multiset

SCNP uses extensions of wellfounded relations to finite sets and multisets.

Given a wellfounded relation $R :: (\alpha \times \alpha)\ set$, we can define a wellfounded relation $\textit{max-ext}\ R :: (\alpha\ set \times \alpha\ set)\ set$, such that a set X is smaller than Y, iff the greatest element of X is smaller than the greatest element of Y. Since a maximal element may not exist and be uniquely defined for arbitrary relations, we instead formulate the definition as follows:

Definition 3.17 (Max Extension)**.**
$\textit{max-ext}\ R =$
$\{(X,\ Y).\ \textit{finite}\ X \wedge \textit{finite}\ Y \wedge Y \neq \emptyset \wedge (\forall x{\in}X.\ \exists y{\in}Y.\ (x,\ y) \in R)\}$

The proof that this extension preserves wellfoundedness is a four-level nested induction, and can be found in the Isabelle library.

Lemma 3.18. $\textit{wf}\ R \Longrightarrow \textit{wf}\ (\textit{max-ext}\ R)$

Similarly, we define the min extension of a relation. Compared to max, the order of quantification is reversed, and surprisingly we need no finiteness conditions. The intuitive reason is that infinite sets always have minimal elements, but not maximal ones.

Definition 3.19 (Min Extension)**.**
$\textit{min-ext}\ R = \{(X,\ Y)\ |X\ Y.\ X \neq \emptyset \wedge (\forall y{\in}Y.\ \exists x{\in}X.\ (x,\ y) \in R)\}$

Lemma 3.20. $\textit{wf}\ R \Longrightarrow \textit{wf}\ (\textit{min-ext}\ R)$

The third extension is the multiset extension, which is well-known from the literature, and its formalization is already present in Isabelle. Here a relation R is lifted to multisets instead of sets:

Definition 3.21 (Multiset Extension)**.**
$\textit{mult1}\ R =$
$\{(N,\ M).$
$\exists\ a\ M0\ K.$
$\quad M = M0 + \{\!|a|\!\} \wedge N = M0 + K \wedge (\forall\ b.\ b \in\!\#\ K \longrightarrow (b,\ a) \in R)\}$
$\textit{ms-ext}\ R = (\textit{mult1}\ R)^{+}$

Lemma 3.22. $\textit{wf}\ R \Longrightarrow \textit{wf}\ (\textit{ms-ext}\ R)$

We refer to Baader and Nipkow [7] for a discussion of the multiset order.

Note that $(X,\ Y) \in \textit{max-ext}\ R$ implies $(\textit{mset}\ X,\ \textit{mset}\ Y) \in \textit{ms-ext}\ R$, where *mset* coerces sets to multisets. It seems that one would never need the max-extension in a termination proof, since the multiset extension is stronger. But the picture changes when we look at the non-strict counterparts. For example, $\{2, 1, 1\} \leq \{2, 1\}$ when compared with the max-order, but $\{\!\{2, 1, 1\}\!\} > \{\!\{2, 1\}\!\}$ in the multiset order.

Ben-Amram and Codish also use a fourth order they call the *dual multiset order*, but experiments show that it contributes very little to the power of the approach, and we decided not to use it, since proof reconstruction for it is technically more involved. This is because the dual multiset order is only wellfounded for a fixed bound on the number of elements in the multiset.

Level mappings

Instead of measure functions into *nat*, SCNP uses so-called *level mappings*:

$$lev :: \tau \Rightarrow (nat \times nat)\ set/multiset$$

These functions map an argument $x :: \tau$ into a set or multiset of pairs. The first component of such a pair is a measure function applied to x, and the second component is a fixed natural number which is called a *tag*. The pairs are ordered lexicographically, and this order on pairs is now lifted using one of the extensions *max-ext*, *min-ext*, or *ms-ext*.

Example 3.23. The termination proof for the function *perm* above works with the multiset ordering and the level mapping

$$\lambda(m, n, r).\ \{\!\{(m, 0), (n, 0), (r, 0)\}\!\}.$$

That is, we map each triple to a multiset of three components. The tags are always zero here (which means that they play no role in this example). With this ordering, both calls are strictly *ms*-decreasing.

If the size-change problem has multiple program points, a level mapping on the sum type is composed from level mappings on the individual types using *sum-case*.

It is easy to see that SCNP subsumes the measure combination approach we described in the previous section for mutual recursive functions: An interpretation that assigns a simple measure to each of the functions corresponds to singleton sets, which can be compared with, e.g., the max-extension. Ben-Amram and Codish prove that it is also NP-complete [11].

3.8.3 SAT encoding

Our SCNP solver uses the SAT encoding described by Ben-Amram and Codish. This encoding produces a formula which contains additional constraints of the form $a < b$ or $a \leq b$, where a and b are special variables ranging over natural numbers up to a fixed bound. These constraints encode the relationships between the tags, and are easily compiled to plain SAT using binary encodings [30]. Table 3.1 lists the different kinds of propositional variables used in the SAT encoding, together with their intended meaning.

Given a set of calls $\mathcal{C}$, we produce the following constraints: First, we expect from a solution that all graphs are weakly decreasing and at least one call is strictly decreasing:

$e^C_{i>j}$	Expresses that in call C, the we have a strict descent from the ith to the jth data position, taking both the information in the size-change graph and the tags into account.
$e^C_{i\geq j}$	The same, but for weak descent.
s^p_i	Indicates that the data position i is selected for the (multi)set of control point p.
strict^C	Indicates that call C is strictly decreasing.
weak^C	Indicates that call C is weakly decreasing.
$\gamma^C_{i,j}$	Indicates that position i is used to "cover" position j in the multiset order. It may either a) cover multiple positions strictly, or b) cover a single position weakly. (unused for max and min order)
ε^C_i	Indicates that case a) above is true for position i. (unused for max and min order)
tag^p_i	Numerical variable; represents the tag associated with control point p and data position i.

Table 3.1: Variables for encoding SCNP into SAT.

Descent constraints:

$$\bigwedge_{C\in\mathcal{C}} \mathit{weak}^C \quad\wedge\quad \bigvee_{C\in\mathcal{C}} \mathit{strict}^C$$

The results of the local descent proofs, visualized by the size-change graphs, are encoded by the following formula, for each $C \in \mathcal{C}$, going from program point p to q, where there are n data positions for p and m for q.

Graph constraints:

$$\bigwedge_{\substack{1\leq i\leq n\\ 1\leq j\leq m}} \left(\begin{array}{cl} & (e^C_{i>j} \leftrightarrow (M^C_{ij} = <) \vee (M^C_{ij} = \leq) \wedge (\mathit{tag}^p_i > \mathit{tag}^q_j)) \\ \wedge & (e^C_{i\geq j} \leftrightarrow (M^C_{ij} = <) \vee (M^C_{ij} = \leq) \wedge (\mathit{tag}^p_i \geq \mathit{tag}^q_j)) \end{array} \right)$$

The remaining constraints further describe the variables strict^C and weak^C, by encoding the requirements of the *max*-, *min*- or *ms* orders. We add one of the following constraint sets, depending on what order we are searching for.

The constraints for the max and min orders directly correspond to the definitions of the respective orders (Defs. 3.17 and 3.19), where the quantifiers are represented as finite conjunctions and disjunctions.

Max order constraints:

$$\mathit{weak}^C \leftrightarrow \bigwedge_{1\leq j\leq m} \left(s^q_j \rightarrow \bigvee_{1\leq i\leq n} (s^p_i \wedge e^C_{i\geq j}) \right)$$

$$\mathit{strict}^C \leftrightarrow \bigwedge_{1\leq j\leq m} \left(s^q_j \rightarrow \bigvee_{1\leq i\leq n} (s^p_i \wedge e^C_{i>j}) \right) \wedge \bigvee_{1\leq i\leq n} s^p_i$$

Min order constraints:

$$\begin{aligned}
\mathit{weak}^C &\leftrightarrow \bigwedge_{1\le i\le n}\left(s_i^p \to \bigvee_{1\le j\le m}(s_j^q \wedge e_{i\ge j}^C)\right)\\
\mathit{strict}^C &\leftrightarrow \bigwedge_{1\le i\le n}\left(s_i^p \to \bigvee_{1\le j\le m}(s_j^q \wedge e_{i> j}^C)\right) \wedge \bigvee_{1\le j\le m} s_j^q
\end{aligned}$$

The encoding of the multiset order uses the following idea: If $(X,\ Y) \in$ *ms-ext* R, then each element of X must be "covered" by an element of Y. An element y may either cover multiple elements x if $y > x$, or it may cover *one* element such that $y \ge x$. In the formula below, the operator $\oplus$ abbreviates that exactly one of the set of variables should be true.

Multiset order constraints:

$$\begin{aligned}
\mathit{weak}^C &\leftrightarrow \bigwedge_{1\le j\le m}(s_j^q \to \bigvee_{1\le i\le n}\gamma_{i,j}^C)\\
\mathit{strict}^C &\leftrightarrow \bigvee_{1\le i\le n}(s_i^p \wedge \neg\varepsilon_i^C)
\end{aligned}$$

$$\bigwedge_{\substack{1\le i\le n\\1\le j\le m}} \gamma_{i,j}^C \to s_i^p \wedge s_j^q \wedge e_{i\ge j}^C \wedge (\neg\varepsilon_i^C \leftrightarrow e_{i>j}^C)$$

$$\bigwedge_{1\le i\le n} s_i^p \to \varepsilon_i^C \to \oplus\{\gamma_{i,j}^C | 1 \le j \le m\}$$

Collecting all the constraints, we obtain a SAT instance that we can pass to an off-the-shelf SAT solver. Thanks to the work of Weber [112], Isabelle already provides an interface to several high-performance SAT-solvers, including zChaff [76] and MiniSAT [36]. In our setup, the SAT solver is invoked multiple times, once for each order.

3.8.4 Proof reconstruction

From a satisfying assignment we obtain from the SAT solver, we can directly extract the necessary information that serves as a certificate. The proof reconstruction requires:

1. The type of extension (max, min or ms) for which we found a solution.
2. The level mapping, consisting of the (multi)set for each program point. We can read this from the values of the s_p^i-variables and the tags.
3. The calls that are strictly decreasing (strict^C). These calls will be removed from the termination goal.
4. A *covering function* for each call, which maps argument positions to other argument positions, they cover (or are covered by).

Put differently, the information in the certificate contains all the instances for the "existential quantifiers" in the propositional encoding.

Definition of the relations

The following relations play a role in the proof reconstruction:

$$\begin{array}{ll} \mathit{lex}_< & = \{((a,\, b),\, (c,\, d))\ | a\ b\ c\ d.\ a < c \lor a \le c \land b < d\} \\ \mathit{lex}_\le & = \mathit{lex}_< \cup \mathit{Id} \\ \mathit{max}_< & = \mathit{max\text{-}ext}\ \mathit{lex}_< \\ \mathit{max}_\le & = \mathit{max\text{-}ext}\ \mathit{lex}_\le \cup \{(\emptyset,\, \emptyset)\} \\ \mathit{min}_< & = \mathit{min\text{-}ext}\ \mathit{lex}_< \\ \mathit{min}_\le & = \mathit{min\text{-}ext}\ \mathit{lex}_\le \cup \{(\emptyset,\, \emptyset)\} \\ \mathit{ms}_< & = \mathit{ms\text{-}ext}\ \mathit{lex}_< \\ \mathit{ms}_\le & = \mathit{ms}_< \cup \mathit{Id} \end{array}$$

It is not hard to show that $(\mathit{max}_<,\, \mathit{max}_\le)$, $(\mathit{min}_<,\, \mathit{min}_\le)$, and $(\mathit{ms}_<,\, \mathit{ms}_\le)$ are reduction pairs.

We now take the inverse image of one of these reduction pairs under the level mapping. Hence we must show that the individual calls are strictly or weakly decreasing.

Proving max- or min-descent

Proving that two given sets are max-decreasing amounts to showing that for each $x \in X$ there is a $y \in Y$ such that $x < y$. The mapping from x to y is part of the certificate, so proof reconstruction merely consists of applying the following introduction rules in a controlled manner:

$$\frac{\mathit{finite}\ Y \qquad Y \neq \emptyset}{(\emptyset,\, Y) \in \mathit{max}_<} \qquad \frac{y \in Y \qquad (x,\, y) \in \mathit{lex}_< \qquad (X,\, Y) \in \mathit{max}_<}{(\{x\} \cup X,\, Y) \in \mathit{max}_<}$$

In a concrete goal, the sets X and Y are always enumerations of the form $\{x,\, y,\, z,\, \ldots\}$, and the rules above can be applied in a syntax-directed way. At each application of the rule for nonempty sets, the variable y must be instantiated with an element of Y, which is selected by the certificate. Hence, the subgoals *finite* X and $X \neq \emptyset$ are easily discharged, since X is always a concrete set.

Proofs for weak descent and for the *min* order are similar using analogous rules given in Fig. 3.4.

Proving multiset descent

Multiset descent can be reduced to max-descent as follows. If $(A,\, B) \in \mathit{ms}_<$, then A and B can be expressed as $Z + A'$ and $Z + B'$, where $(A',\, B') \in \mathit{max}_<$.

Since the size-change abstraction uses only inequalities, we cannot express that the Z-parts on both sides are actually equal. However, it is sufficient if they satisfy the following relation, which is the pairwise lifting of $\mathit{lex}_\le$:

$$\frac{}{(\{\!\!\{\}\!\!\},\, \{\!\!\{\}\!\!\}) \in \mathit{pairwise}_\le} \qquad \frac{(x,\, y) \in \mathit{lex}_\le \qquad (X,\, Y) \in \mathit{pairwise}_\le}{(\{\!\!\{x\}\!\!\} + X,\, \{\!\!\{y\}\!\!\} + Y) \in \mathit{pairwise}_\le}$$

This gives us the following introduction rule for strict multiset-descent, where the function *set-of* converts multisets to plain sets:

$$\frac{(Z,\, Z') \in \mathit{pairwise}_\le \qquad (\mathit{set\text{-}of}\ A,\, \mathit{set\text{-}of}\ B) \in \mathit{max}_<}{(Z + A,\, Z' + B) \in \mathit{ms}_<}$$

Rules for $max_<$:

$$\frac{\textit{finite } Y \qquad Y \neq \emptyset}{(\emptyset,\ Y) \in max_<} \qquad \frac{y \in Y \qquad (x,\ y) \in lex_< \qquad (X,\ Y) \in max_<}{(\{x\} \cup X,\ Y) \in max_<}$$

Rules for $max_\leq$:

$$\frac{\textit{finite } X}{(\emptyset,\ X) \in max_\leq} \qquad \frac{y \in YS \qquad (x,\ y) \in lex_\leq \qquad (XS,\ YS) \in max_\leq}{(\{x\} \cup XS,\ YS) \in max_\leq}$$

Rules for $min_<$:

$$\frac{X \neq \emptyset}{(X,\ \emptyset) \in min_<} \qquad \frac{x \in XS \qquad (x,\ y) \in lex_< \qquad (XS,\ YS) \in min_<}{(XS,\ \{y\} \cup YS) \in min_<}$$

Rules for $min_\leq$:

$$\frac{}{(X,\ \emptyset) \in min_\leq} \qquad \frac{x \in XS \qquad (x,\ y) \in lex_\leq \qquad (XS,\ YS) \in min_\leq}{(XS,\ \{y\} \cup YS) \in min_\leq}$$

Rules for $ms_<$:

$$\frac{(Z,\ Z') \in pairwise_\leq \qquad (\textit{set-of } A,\ \textit{set-of } B) \in max_<}{(Z + A,\ Z' + B) \in ms_<}$$

Rules for $ms_\leq$:

$$\frac{(Z,\ Z') \in pairwise_\leq \qquad (\textit{set-of } A,\ \textit{set-of } B) \in max_<}{(Z + A,\ Z' + B) \in ms_\leq}$$

$$\frac{(Z,\ Z') \in pairwise_\leq}{(Z + \{\!|\,|\!\},\ Z' + \{\!|\,|\!\}) \in ms_\leq}$$

Rules for $pairwise_\leq$:

$$\frac{}{(\{\!|\,|\!\},\ \{\!|\,|\!\}) \in pairwise_\leq} \qquad \frac{(x,\ y) \in lex_\leq}{(\{\!|x|\!\},\ \{\!|y|\!\}) \in pairwise_\leq}$$

$$\frac{(x,\ y) \in lex_\leq \qquad (X,\ Y) \in pairwise_\leq}{(\{\!|x|\!\} + X,\ \{\!|y|\!\} + Y) \in pairwise_\leq}$$

Rules for $lex_<$ and $lex_\leq$:

$$\frac{a < b}{((a,\ s),\ (b,\ t)) \in lex_<} \qquad \frac{a \leq b \qquad s < t}{((a,\ s),\ (b,\ t)) \in lex_<}$$

$$\frac{a < b}{((a,\ s),\ (b,\ t)) \in lex_\leq} \qquad \frac{a \leq b \qquad s \leq t}{((a,\ s),\ (b,\ t)) \in lex_\leq}$$

Figure 3.4: Introduction rules for SCNP proof reconstruction

The information in the certificate tells us exactly how the multisets decompose in the two parts. Before applying the introduction rule above, we must rewrite the goal using associativity and commutativity, to bring the element of the multiset in the right order.

Reconstruction for weak multiset order is similar. The rules are given in Fig. 3.4.

Example 3.24. For the function *perm*, we show that the first call is strictly ms-decreasing:

$$\begin{array}{l}1.\ 0 < r \Longrightarrow \\ \quad (\{\!\!\{(m,\ 0),\ (r - 1,\ 0),\ (n,\ 0)\}\!\!\},\ \{\!\!\{(m,\ 0),\ (r,\ 0),\ (n,\ 0)\}\!\!\}) \\ \quad \in ms_{<}\end{array}$$

We rewrite the multisets using associativity and commutativiy to bring them in the form to apply the introduction rule for $ms_{<}$:

$$\begin{array}{l}1.\ 0 < r \Longrightarrow \\ \quad (\{\!\!\{(m,\ 0),\ (n,\ 0)\}\!\!\} + \{\!\!\{(r - 1,\ 0)\}\!\!\}, \\ \quad\ \{\!\!\{(m,\ 0),\ (n,\ 0)\}\!\!\} + \{\!\!\{(r,\ 0)\}\!\!\}) \\ \quad \in ms_{<}\end{array}$$

After applying the induction rule, we get two subgoals:

$$\begin{array}{l}1.\ 0 < r \Longrightarrow \\ \quad (\{\!\!\{(m,\ 0),\ (n,\ 0)\}\!\!\},\ \{\!\!\{(m,\ 0),\ (n,\ 0)\}\!\!\}) \in pairwise_{\leq} \\ 2.\ 0 < r \Longrightarrow (set\text{-}of\ \{\!\!\{(r - 1,\ 0)\}\!\!\},\ set\text{-}of\ \{\!\!\{(r,\ 0)\}\!\!\}) \in max_{<}\end{array}$$

We apply the introduction rules for $pairwise_{\leq}$ and unfold *set-of*:

$$\begin{array}{l}1.\ 0 < r \Longrightarrow ((m,\ 0),\ (m,\ 0)) \in lex_{\leq} \\ 2.\ 0 < r \Longrightarrow ((n,\ 0),\ (n,\ 0)) \in lex_{\leq} \\ 3.\ 0 < r \Longrightarrow (\{(r - 1,\ 0)\},\ \{(r,\ 0)\}) \in max_{<}\end{array}$$

Then we apply the introduction rules for $lex_{\leq}$:

$$\begin{array}{l}1.\ 0 < r \Longrightarrow m \leq m \\ 2.\ 0 < r \Longrightarrow 0 \leq 0 \\ 3.\ 0 < r \Longrightarrow n \leq n \\ 4.\ 0 < r \Longrightarrow 0 \leq 0 \\ 5.\ 0 < r \Longrightarrow (\{(r - 1,\ 0)\},\ \{(r,\ 0)\}) \in max_{<}\end{array}$$

Now, subgoals 1 and 3 are the local descent properties that we have already proved. Subgoals 2 and 4 are inequalities between number literals and are discharged using *arith*. For the last subgoal, we apply the second introduction rule for $max_{<}$, instantiating y with $(r,\ 0)$:

$$\begin{array}{l}1.\ 0 < r \Longrightarrow ((r - 1,\ 0),\ r,\ 0) \in lex_{<} \\ 2.\ 0 < r \Longrightarrow (\emptyset,\ \{(r,\ 0)\}) \in max_{<}\end{array}$$

Then, using the introduction rule for $lex_{<}$, we can apply a local descent property again. The last subgoal is solved with the first introduction rule for $max_{<}$, whose premises are trivial.

Fortunately, these tedious proofs are fully automated in a tactic.

3.9 Implementation and Practical Considerations

3.9.1 Strategies

It is easy to see that the little calculus we have introduced is confluent: The three rules for reduction pairs, decomposition, and trivial calls can be applied in any order. In particular, after removing some calls using reduction pairs, the dependency graph may fall apart into smaller SCCs again. Whenever different reduction pairs are applicable (possibly removing different calls), we can make an arbitrary choice, and the other reduction pairs will still be applicable in the next step. So it is always safe to use any applicable rule, and we need not backtrack if we get stuck. The calculus is also terminating, since every rule application makes the goal smaller[3].

We implemented two strategies, which are now available as proof methods in Isabelle: The method *lexicographic-order* implements the basic approach presented in §3.5, which basically constructs lexicographic combinations of measure functions. In particular, it does not use graph decomposition or SCNP.

The dependency graph decomposition (§3.6) and SCNP approach (§3.8) are combined in a single method with the brand name *sizechange*, although the cryptic *dp-scnp* would be technically more correct. The *sizechange* method is strictly more powerful than *lexicographic-order*.

Performance While the order in which the different methods are applied has no influence on the power of the overall termination prover, it can make a difference in performance. The following considerations turned out to be useful to minimize the time spent on finding the proof:

- In a call from a function to itself (as opposed to another function in a mutual recursion), the local descent matrix entries outside the diagonal will only be interesting for functions with permuted arguments. Since these functions are currently rare, it can save time to omit the derivation of these entries in a first round and try to find a termination proof with the rest of the information. If that fails, the procedure can be repeated with the full information. This can save many calls to *auto*, which is potentially expensive.

- The SAT solving for SCNP is notably faster (we measured a factor of five) if no tags are used. However only a few problems require tags. Thus tags are turned off in the first round.

- For problems that do not need graph decomposition, deriving the dependency graph is just wasted time. Thus, the required theorems are only proved after the first round of SCNP got stuck.

However, this kind of tuning is based on relatively little experimental data, together with a lot of guesswork, and the current setup should be revisited when more experience with the *sizechange* method is available.

[3]Since this is a chapter on termination, here is the rigorous argument: A termination goal state consists of multiple independent subgoals, each consisting of a set of calls. Assign to each subgoal the number of its calls, and collect these counters in a multiset. Now each rule leads to a smaller proof state with respect to the multiset order.

3.9.2 Examples

Ackermann function

The Ackermann function, defined by the equations

$$\begin{aligned}
&\mathit{ack}\ 0\ m &&= \mathit{Suc}\ m\\
&\mathit{ack}\ (\mathit{Suc}\ n)\ 0 &&= \mathit{ack}\ n\ 1\\
&\mathit{ack}\ (\mathit{Suc}\ n)\ (\mathit{Suc}\ m) &&= \mathit{ack}\ n\ (\mathit{ack}\ (\mathit{Suc}\ n)\ m)
\end{aligned}$$

is easily proved total by *lexicographic-order*.

Many parameters

Both HOL4 and HOL Light fail to prove termination of the following function in reasonable time, because they enumerate all possible lexicographic combinations of parameters, which is impracticable for functions with many parameters. In contrast, *lexicographic-order* succeeds within a second, since it solves one call at a time, with no exponential explosion.

$$\begin{aligned}
&\mathit{blowup}\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 &&= 0\\
&\mathit{blowup}\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ (\mathit{Suc}\ i) &&= \mathit{Suc}\ (\mathit{blowup}\ i\ i\ i\ i\ i\ i\ i\ i\ i)\\
&\mathit{blowup}\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ (\mathit{Suc}\ h)\ i &&= \mathit{Suc}\ (\mathit{blowup}\ h\ h\ h\ h\ h\ h\ h\ h\ i)\\
&\mathit{blowup}\ 0\ 0\ 0\ 0\ 0\ 0\ (\mathit{Suc}\ g)\ h\ i &&= \mathit{Suc}\ (\mathit{blowup}\ g\ g\ g\ g\ g\ g\ g\ h\ i)\\
&\mathit{blowup}\ 0\ 0\ 0\ 0\ 0\ (\mathit{Suc}\ f)\ g\ h\ i &&= \mathit{Suc}\ (\mathit{blowup}\ f\ f\ f\ f\ f\ f\ g\ h\ i)\\
&\mathit{blowup}\ 0\ 0\ 0\ 0\ (\mathit{Suc}\ e)\ f\ g\ h\ i &&= \mathit{Suc}\ (\mathit{blowup}\ e\ e\ e\ e\ e\ f\ g\ h\ i)\\
&\mathit{blowup}\ 0\ 0\ 0\ (\mathit{Suc}\ d)\ e\ f\ g\ h\ i &&= \mathit{Suc}\ (\mathit{blowup}\ d\ d\ d\ d\ e\ f\ g\ h\ i)\\
&\mathit{blowup}\ 0\ 0\ (\mathit{Suc}\ c)\ d\ e\ f\ g\ h\ i &&= \mathit{Suc}\ (\mathit{blowup}\ c\ c\ c\ d\ e\ f\ g\ h\ i)\\
&\mathit{blowup}\ 0\ (\mathit{Suc}\ b)\ c\ d\ e\ f\ g\ h\ i &&= \mathit{Suc}\ (\mathit{blowup}\ b\ b\ c\ d\ e\ f\ g\ h\ i)\\
&\mathit{blowup}\ (\mathit{Suc}\ a)\ b\ c\ d\ e\ f\ g\ h\ i &&= \mathit{Suc}\ (\mathit{blowup}\ a\ b\ c\ d\ e\ f\ g\ h\ i)
\end{aligned}$$

Multiplication by shifting and addition

Pandya and Joseph [86] introduced a new proof rule for total correctness of mutually recursive procedures. The contribution of this proof rule is a refined method for proving termination by analysing the procedure call graph. They motivate their approach with an imperative version of the following example:

$$\begin{aligned}
&\mathit{prod}\ x\ y\ z &&= \textsf{if}\ y\ \mathit{mod}\ 2 = 0\ \textsf{then}\ \mathit{eprod}\ x\ y\ z\ \textsf{else}\ \mathit{oprod}\ x\ y\ z\\
&\mathit{oprod}\ x\ y\ z &&= \mathit{eprod}\ x\ (y - 1)\ (z + x)\\
&\mathit{eprod}\ x\ y\ z &&= \textsf{if}\ y = 0\ \textsf{then}\ z\ \textsf{else}\ \mathit{prod}\ (2 * x)\ (y\ \mathit{div}\ 2)\ z
\end{aligned}$$

In the calls from *oprod* and *eprod* the second argument decreases but in the calls from *prod* all arguments are unchanged. The termination proof in Isabelle first removes the decreasing calls. Then the dependency graph has no cycles any longer.

Pedal and Coast

Homeier and Martin [54] describe a nontrivial graph analysis, and their main example is an imperative version of what they call the *bicycling* program:

$$
\begin{array}{ll}
\mathit{pedal} & :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \\
\mathit{coast} & :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow \mathit{nat} \\
\\
\mathit{pedal}\ 0\ m\ c & = c \\
\mathit{pedal}\ n\ 0\ c & = c \\
\mathit{pedal}\ (\mathit{Suc}\ n)\ (\mathit{Suc}\ m)\ c & = \textsf{if}\ n < m \\
 & \quad \textsf{then}\ \mathit{coast}\ n\ m\ (c + \mathit{Suc}\ m) \\
 & \quad \textsf{else}\ \mathit{pedal}\ n\ (\mathit{Suc}\ m)\ (c + \mathit{Suc}\ m) \\
\mathit{coast}\ n\ m\ c & = \textsf{if}\ n < m \\
 & \quad \textsf{then}\ \mathit{coast}\ n\ (m - 1)\ (c + n) \\
 & \quad \textsf{else}\ \mathit{pedal}\ n\ m\ (c + n)
\end{array}
$$

They claim that termination would be difficult to prove using the rule by Pandya and Joseph. In our framework, the problem is no more difficult. Again it is solved using dependency graph decomposition.

Polynomial addition

The following example comes from a formalization of a descision procedure for equalities in a commutative ring, adapted from similar work in Coq [46] (the Isabelle version was done by Bernhard Häupler). The function adds two polynomials, represented by a datatype with three constructors Pc, $Pinj$ and PX, whose meaning need not concern us here:

$$\mathit{add}\ (Pc\ a)\ (Pc\ b) = Pc\ (a + b)$$

$$\mathit{add}\ (Pc\ c)\ (Pinj\ i\ P) = Pinj\ i\ (\mathit{add}\ P\ (Pc\ c))$$

$$\mathit{add}\ (Pc\ c)\ (PX\ P\ i\ Q) = PX\ P\ i\ (\mathit{add}\ Q\ (Pc\ c))$$

$$
\begin{array}{l}
\mathit{add}\ (Pinj\ x\ P)\ (Pinj\ y\ Q) = \\
(\textsf{if}\ x = y\ \textsf{then}\ \mathit{mkPinj}\ x\ (\mathit{add}\ P\ Q) \\
\ \textsf{else}\ \textsf{if}\ y < x\ \textsf{then}\ \mathit{mkPinj}\ y\ (\mathit{add}\ (Pinj\ (x - y)\ P)\ Q) \\
\qquad \textsf{else}\ \underline{\mathit{add}\ (Pinj\ y\ Q)\ (Pinj\ x\ P)})
\end{array}
$$

$$
\begin{array}{l}
\mathit{add}\ (Pinj\ x\ P)\ (PX\ Q\ y\ R) = \\
(\textsf{if}\ x = 0\ \textsf{then}\ \mathit{add}\ P\ (PX\ Q\ y\ R) \\
\ \textsf{else}\ \textsf{if}\ x = 1\ \textsf{then}\ PX\ Q\ y\ (\mathit{add}\ P\ R) \\
\qquad \textsf{else}\ PX\ Q\ y\ (\mathit{add}\ (Pinj\ (x - 1)\ P)\ R))
\end{array}
$$

$$
\begin{array}{l}
\mathit{add}\ (PX\ P_1\ x\ P_2)\ (PX\ Q_1\ y\ Q_2) = \\
(\textsf{if}\ x = y\ \textsf{then}\ \mathit{mkPX}\ (\mathit{add}\ P_1\ Q_1)\ x\ (\mathit{add}\ P_2\ Q_2) \\
\ \textsf{else}\ \textsf{if}\ y < x \\
\qquad \textsf{then}\ \mathit{mkPX}\ (\mathit{add}\ (PX\ P_1\ (x - y)\ (Pc\ 0))\ Q_1)\ y\ (\mathit{add}\ P_2\ Q_2) \\
\qquad \textsf{else}\ \underline{\mathit{add}\ (PX\ Q_1\ y\ Q_2)\ (PX\ P_1\ x\ P_2)})
\end{array}
$$

$$\mathit{add}\ (Pinj\ i\ P)\ (Pc\ c) = \underline{\mathit{add}\ (Pc\ c)\ (Pinj\ i\ P)}$$

$$\mathit{add}\ (PX\ P\ i\ Q)\ (Pc\ c) = \underline{\mathit{add}\ (Pc\ c)\ (PX\ P\ i\ Q)}$$

$$\mathit{add}\ (PX\ Q\ y\ R)\ (Pinj\ x\ P) = \underline{\mathit{add}\ (Pinj\ x\ P)\ (PX\ Q\ y\ R)}$$

In the underlined cases the function just calls itself with permuted arguments. This avoids duplicating the code from other clauses — a sensible programming pattern for commutative functions. However, without an analysis dealing with argument permutation, it would be hard to get the definition accepted by Isabelle, since even specifying a termination order manually is nontrivial. In the original version, the definition contains the duplicated cases instead, to avoid this problem.

Note that such duplication does not only concern the function specification, but will turn up again in induction proofs about the function, as the induction rule is generated from the definition. This leads to redundant cases, whose analogous proofs have to be copy-and-pasted.

Our implementation proves termination automatically, using the multiset order and graph decomposition.

3.9.3 Evaluation

To evaluate our approach, we tested our termination prover on all non-primitive recursive function definitions in the Isabelle Distribution and the Archive of Formal Proofs [1]. The results can be summarized as follows:

1. The large majority (87%) of the termination problems can be solved with the simpler method *lexicographic-order*.

2. Using the more refined analyses of the *sizechange* method, only a handful of additional problems can be solved.

First of all, this is a big success, given the fact that previously all termination proofs had to be done manually. A possible explanation for the fact that *sizechange* cannot show its strengths on the available data could be that users, knowing about Isabelle's limitations in that area, tried to avoid function definitions that would require a difficult manual termination proof, and used other modeling techniques like inductive relations instead. It remains to see whether this changes now that the method is generally available in Isabelle. But the *add* function discussed above already shows the potential of size-change termination. The examples where the automated termination proof fails mainly fall in one of the following categories:

1. Definitions which use a customized size function, where some constructors are weighted more than others. This method is essentially polynomial interpretation, done manually.

2. Functions over naturals or integers, where the argument is increasing but bounded from above.

3. "Difficult" functions, where a domain specific semantic argument is used for termination. The unification algorithm from §2.7.3 is such an example.

4. Functions over more powerful set theoretic constructions like ordinals.

5. Functions that are not total and require special treatment anyway.

3.9.4 Feedback from failure

If our analysis fails to find a termination proof, this can have several reasons:

1. The function is indeed non-terminating.
2. The function is terminating, but the termination argument is not captured by the measure functions used or the ways they are combined.
3. Some local descent proof did not go through although the corresponding property is true. The reason for that can be an inherent weakness of the automation employed or simply a missing lemma.

Of course it is impossible for the system to distinguish between these three classes of errors. However, instead of just failing, we can provide valuable information to the user by showing the matrices which summarize the local descent proofs. The user can then compare the results with his expectations. For unsuccessful attempts, printing the unfinished proof states can give feedback on lemmas that may be necessary to complete the proof. This is a useful debugging aid, but requires a basic understanding of how the termination prover works.

3.10 Full Size-Change Termination

The SCNP approach produces nice certificates, but it cannot handle all instances of SCT. In this section we describe how the size-change principle in its full generality can be used for termination proofs in Isabelle. Unlike the certificate-based approach of SCNP, we use a reflected decision procedure, whose correctness is formalized within the logic itself. Using tactics, a concrete termination problem can be automatically converted to a size-change instance, modelled as a graph inside HOL, which serves as input to the decision procedure.

The approach presented in this section has not been integrated into Isabelle, although the formalization is available in the distribution. Integrating it seems not worth the effort for three resons:

- It is an all-or-nothing approach that either solves a termination goal completely or fails, but never simplifies it, which makes it less modular than the other methods.
- The current decision procedure on graphs is only an inefficient prototype, which scales very poorly. This could be fixed by implementing and verifying a procedure using better data structures.
- Most importantly, all practical termination problems we encountered so far that are solved by SCT are also solved by SCNP. We can construct problems that require foll SCT, but they do not occur in user theories we have seen.

Nevertheless we find it worthwhile to present the full criterion, since the reflection-based approach differs considerably from the other ones, and since the formalization of the criterion is interesting in its own right.

3.10.1 Formalization

Kleene algebras

The core of SCT checking is the computation of a transitive closure, so we start by defining an axiomatic type class of Kleene algebras, which provide the most general structure for such an operation. With this approach, the formulation of the algorithm is kept separate from the concrete data structures, which allows very abstract reasoning using just simple algebraic laws. Since our graphs will be special Kleene algebras, the corresponding theorems simply follow as instances.

Following the axiomatization by Kozen [62], Kleene algebras are idempotent semirings with an order defined as $(a \le b) = (a + b = b)$. Additionally, they include a star-operation satisfying the following four laws:

$$\begin{array}{ll} \mathit{1} + aa^* \le a^* & \qquad ax \le x \Longrightarrow a^*x \le x \\ \mathit{1} + a^*a \le a^* & \qquad xa \le x \Longrightarrow xa^* \le x \end{array}$$

These axioms follow from a stronger property, called *-continuity:

$$ab^*c = (\mathit{SUP}\ n.\ ab^nc)$$

where b^n denotes iterated multiplication. We define transitive closures as $a^+ = a^*a$.

In the literature, general transitive closures are often described using closed semirings. Kleene algebras are slightly more general: Every closed semiring is a *-continuous Kleene algebra. However, our formalization does not use this generality, as our instances are closed semirings as well.

Graphs

We represent directed edge-labeled graphs as ternary relations. Hence, graphs may have self-edges, and between two nodes there may be several edges:

$$\textbf{datatype}\ (\alpha,\ \beta)\ \mathit{graph} = \mathit{Graph}\ ((\alpha \times \beta \times \alpha)\ \mathit{set})$$

Instead of using relations directly, we wrap graphs into their own type constructor. This allows us to use type classes to overload common notation for graph composition (written as multiplication), exponentiation and transitive closure. We write $x \xrightarrow[G]{e} y$ if G has an edge between nodes x and y, which is labeled with e. If we do not care about the label, we just write $x \xrightarrow[G]{} y$.

If the type of the edges has a multiplication and unit operation, these can be lifted to graphs, preserving the monoid structure:

$$\begin{array}{lcl} p \xrightarrow[GH]{b} q & = & \exists\, k\ e\ e'.\ p \xrightarrow[G]{e} k \xrightarrow[H]{e'} q \wedge b = ee' \\ p \xrightarrow[1]{b} q & = & p = q \wedge b = \mathit{1} \end{array}$$

With addition defined as set union, we get a semiring structure with additive and multiplicative identity. Moreover, by taking union and intersection for supremum and infimum, graphs form a complete lattice and we can define the star operation as $G^* = (\mathit{SUP}\ n.\ G^n)$. It is then not hard to show that graphs form a (*-continuous) Kleene algebra.

Paths

We represent infinite paths as sequences of node-edge pairs, using the following type abbreviations:

types
α *sequence* $= nat \Rightarrow \alpha$
(α, β) *ipath* $= (\alpha \times \beta)$ *sequence*

The paths of a graph G are characterized by the predicate *has-ipath*:

$$\textit{has-ipath} \; :: \; (\alpha, \beta) \; graph \Rightarrow (\alpha, \beta) \; ipath \Rightarrow bool$$

$$\textit{has-ipath} \; G \; p = (\forall i. \; fst \; (p \; i) \xrightarrow[G]{snd\ (p\ i)} fst \; (p \; (Suc \; i)))$$

We also need to talk about finite paths and relate them to infinite paths (by taking sub-paths, and constructing infinite paths from finite loops). We omit these details, as they are straightforward.

Size-change graphs

Size-change graphs have two kinds of edges, labeled $<$ and $\leq$. We use the natural numbers as node labels. Control graphs have size-change graphs as their edges.

datatype *scg-edge* $=$ *LESS* $(<)$ | *LEQ* $(\leq)$

types
scg $= (nat, \textit{scg-edge})$ *graph*
control-graph $= (nat, scg)$ *graph*

Given an infinite path in the control graph, a *thread* is a sequence of natural numbers denoting argument positions for every node in the path, such that there are corresponding connected edges. A thread is descending if it contains infinitely many strict edges:

$$\textit{is-desc-thread} \; :: \; nat \; sequence \Rightarrow (nat, scg) \; ipath \Rightarrow bool$$

$$\begin{aligned}&\textit{is-desc-thread} \; \vartheta \; p = \\ &((\exists n. \; \forall i \geq n. \; \vartheta \; i \xrightarrow[snd\ (p\ i)]{} \vartheta \; (Suc \; i)) \wedge \\ &(\exists_\infty i. \; \vartheta \; i \xrightarrow[snd\ (p\ i)]{<} \vartheta \; (Suc \; i)))\end{aligned}$$

Note that threads may also start at a later point in the path. Now the size-change property is defined as

$$SCT \; \mathcal{C} = (\forall p. \; \textit{has-ipath} \; \mathcal{C} \; p \longrightarrow (\exists \vartheta. \; \textit{is-desc-thread} \; \vartheta \; p))$$

The second characterization, which will be proved equivalent, is the basis of the size-change algorithm:

$$SCT' \; \mathcal{C} = (\forall n \; G. \; n \xrightarrow[\mathcal{C}^+]{G} n \wedge GG = G \longrightarrow (\exists p. \; p \xrightarrow[G]{<} p))$$

With these prerequisites we can state the main equivalence result, which corresponds to [66, Thm. 4]:

Theorem 3.25. *finite-cg* $\mathcal{C} \Longrightarrow SCT \; \mathcal{C} = SCT' \; \mathcal{C}$

The condition *finite-cg* $\mathcal{C}$ expresses that the control graph and all its size-change graphs are finite. In the original development it is implicit, since graphs are finite by definition.

The formal proof of Thm. 3.25 can be found in the Isabelle library. It consists of about 1200 lines of Isar proof script, mainly following the informal proof by Lee, Jones and Ben-Amram [66], but with many parts spelled out in much more detail. Like in the informal version, the proof uses Ramsey's Theorem, which is already present in Isabelle's library (the formalization is due to Paulson and Ridge).

3.10.2 Reflecting size-change problems

We must now connect the abstract graph property *SCT* with the termination goals that we are trying to solve. To that aim, we internalize the representation of calls, which makes it accessible to functions defined in HOL.

We define the notion of a *call descriptor*, which is a triple (Γ, r, l) describing a call. All three components have a function type, as they may depend on variables. The function *mk-call* maps a call descriptor to the familiar relation comprehension:

$$\textbf{types}\ (\alpha, \gamma)\ \mathit{call}\ =\ (\gamma \Rightarrow \mathit{bool}) \times (\gamma \Rightarrow \alpha) \times (\gamma \Rightarrow \alpha)$$
$$\mathit{mk\text{-}call}\ ::\ (\alpha, \gamma)\ \mathit{call} \Rightarrow (\alpha \times \alpha)\ \mathit{set}$$
$$\mathit{mk\text{-}call}\ (\Gamma, r, l) = \{(r\ v, l\ v)\ |v.\ \Gamma\ v\}$$

We can easily express calls that have multiple variables by encoding them as a tuple, e.g.,

$$\mathit{mk\text{-}call}\ (\lambda(x, y).\ \Gamma\ x\ y, \lambda(x, y).\ r\ x\ y, \lambda(x, y).\ l\ x\ y) =$$
$$\{(r\ x\ y, l\ x\ y)\ |x\ y.\ \Gamma\ x\ y\}$$

For a list of call descriptors, we define $\mathit{mk\text{-}rel}\ \mathit{cs} = (\bigcup_{c\in \mathit{set}\ \mathit{cs}} \mathit{mk\text{-}call}\ c)$.

Given a termination goal $\mathit{wf}\ (C_1 \cup \ldots \cup C_n)$, we can easily convert it to $\mathit{wf}\ (\mathit{mk\text{-}rel}\ [c_1, \ldots, c_n])$, where $c_1, \ldots, c_n$ are call descriptors. This reification process cannot be defined inside the logic, but must be implemented in a tactic. The only technical challenge is finding a suitable tuple type for γ, which must be a product large enough to contain the variables of each call. The equivalence to the original version simply follows by unfolding the definitions of *mk-rel* and *mk-call*.

To each call (i.e. each control point in the graph), we assign a list of measure functions, which correspond to the data positions. We do this in the usual way, using the procedure from §3.3.1.

The following type abbreviation simplifies the following presentation a little:

types
$\alpha\ \mathit{measure}\ =\ \alpha \Rightarrow \mathit{nat}$

Approximating the control graph

We will now show how to build a size-change problem corresponding to a list of call descriptors. We could follow the same path as in §3.8, where we produced a size-change graph for every call. However, there is another way of constructing a size-change problem, which was first described by Thiemann and Giesl [103] when they applied SCT to term rewriting: It is often beneficial to use the dependency graph as the control graph, since it gives a finer approximation of the control flow. This means that the calls become nodes of the control graph instead of edges.

For the SCNP approach, this was less important since that approach could be combined with dependency graph decomposition. However, the present monolithic technique benefits from this additional precision.

For two call descriptors C_i and C_j, the predicate *no-step* is true if a C_i-call can never be followed by a C_j-call:

$$\begin{aligned}&\textit{no-step} :: (\alpha,\gamma)\ \textit{call} \Rightarrow (\alpha,\gamma)\ \textit{call} \Rightarrow \textit{bool}\\ &\textit{no-step}\ c\ c' = (\textit{mk-call}\ c \circ \textit{mk-call}\ c' = \emptyset)\end{aligned}$$

If we can prove *no-step* c_i c_j, then we can be sure that these calls can never occur in sequence. Otherwise we must add an edge between i and j to our control graph. This edge will carry a size change graph which approximates the size change behaviour of the call.

The predicates $\textit{step}_<$ and $\textit{step}_\leq$ capture strict and non-strict decrease of measures from one call to the next:

$$\begin{aligned}&\textit{step}_< :: (\alpha,\gamma)\ \textit{call} \Rightarrow (\alpha,\gamma)\ \textit{call} \Rightarrow \alpha\ \textit{measure} \Rightarrow \alpha\ \textit{measure} \Rightarrow \textit{bool}\\ &\textit{step}_\leq :: (\alpha,\gamma)\ \textit{call} \Rightarrow (\alpha,\gamma)\ \textit{call} \Rightarrow \alpha\ \textit{measure} \Rightarrow \alpha\ \textit{measure} \Rightarrow \textit{bool}\end{aligned}$$

$$\begin{aligned}&\textit{step}_<\ (\Gamma_1,\ r_1,\ l_1)\ (\Gamma_2,\ r_2,\ l_2)\ m_1\ m_2 =\\ &(\forall\, q_1\ q_2.\ \Gamma_1\ q_1 \wedge \Gamma_2\ q_2 \wedge r_1\ q_1 = l_2\ q_2 \longrightarrow m_2\ (l_2\ q_2) < m_1\ (l_1\ q_1))\end{aligned}$$

$$\begin{aligned}&\textit{step}_\leq\ (\Gamma_1,\ r_1,\ l_1)\ (\Gamma_2,\ r_2,\ l_2)\ m_1\ m_2 =\\ &(\forall\, q_1\ q_2.\ \Gamma_1\ q_1 \wedge \Gamma_2\ q_2 \wedge r_1\ q_1 = l_2\ q_2 \longrightarrow m_2\ (l_2\ q_2) \leq m_1\ (l_1\ q_1))\end{aligned}$$

Now consider a size-change graph G and functions M_1 and M_2 which assign measures to the data positions of C_1 and C_2. We say that G approximates the pair of calls if the claimed inequalities are actually satisfied by the respective measures. This is expressed by the *approx* predicate:

$$\begin{aligned}\textit{approx} :: {}&\textit{scg} \Rightarrow (\alpha,\gamma)\ \textit{call} \Rightarrow (\alpha,\gamma)\ \textit{call}\\ &\Rightarrow (\textit{nat} \Rightarrow \alpha\ \textit{measure}) \Rightarrow (\textit{nat} \Rightarrow \alpha\ \textit{measure}) \Rightarrow \textit{bool}\end{aligned}$$

$$\begin{aligned}&\textit{approx}\ G\ C_1\ C_2\ M_1\ M_2 =\\ &(\forall\, i\, j.\ (i \xrightarrow[G]{<} j \longrightarrow \textit{step}_<\ C_1\ C_2\ (M_1\ i)\ (M_2\ j)) \wedge\\ &\qquad (i \xrightarrow[G]{\leq} j \longrightarrow \textit{step}_\leq\ C_1\ C_2\ (M_1\ i)\ (M_2\ j)))\end{aligned}$$

Now, a control graph $\mathcal{C}$ is a sound description of a given list of call descriptors and measure functions if between any two calls either no step is possible or $\mathcal{C}$ contains the corresponding edge with a size-change graph approximating the call combination[4]:

$$\textit{sound-desc} :: \textit{control-graph} \Rightarrow (\alpha,\gamma)\ \textit{call list} \Rightarrow (\textit{nat} \Rightarrow \alpha\ \textit{measure})\ \textit{list} \Rightarrow \textit{bool}$$

[4]Recall that $xs_{[i]}$ denotes the ith element of the list xs.

$sound\text{-}desc\ \mathcal{A}\ D\ M =$
$(\forall\, n{<}list\text{-}size\ D.$
$\quad \forall\, m{<}list\text{-}size\ D.$
$\quad\quad no\text{-}step\ D_{[n]}\ D_{[m]} \lor (\exists\, G.\ n \xrightarrow[\mathcal{A}]{G} m \land approx\ G\ D_{[n]}\ D_{[m]}\ M_{[n]}\ M_{[m]}))$

Now, it is straigtforward to prove the following:

Theorem 3.26. $sound\text{-}desc\ \mathcal{C}\ D\ M \Longrightarrow SCT\ \mathcal{C} \Longrightarrow wfP\ (mk\text{-}rel\ D)$

With this theorem, which is basically a formal version of the results by Manolios and Vroon [69], we are able to prove wellfoundedness of a relation provided we can express it in terms of a list of call descriptors and find an $\mathcal{C}$ which satisfies SCT and is a sound estimation of the relation.

Building size-change problems

It is not hard to build a custom proof tactic to construct $\mathcal{C}$ and prove $sound\text{-}desc\ \mathcal{C}\ D\ M$:

- For each pair of calls C_i and C_j, try to prove $no\text{-}step\ C_i\ C_j$.
- If this succeeds, no edge needs to be added to $\mathcal{C}$.
- If it fails, construct a size-change graph G, by proving as many of the $step_<$ and $step_\leq$ estimations as possible. For each successful proof, the corresponding edge can be added to the G.

For the "try to prove ..." steps in the above algorithm, we again simply call the method *auto*.

3.10.3 Implementation prototype

Finally, an algorithm for checking the predicate SCT' must be implemented and proved correct. We will present a naive implementation without any optimizations. The prototype limits the performance of our system, but it is sufficient to explain the ideas and demonstrate the overall approach.

We can use Isabelle's code generator to translate the algorithm into ML. The code generator can now also translate definitions involving type classes, which are compiled into dictionaries as it is done in Haskell compilers. Up to Isabelle 2007, the code generation framework also supported the execution of functions involving (finite) sets, which were compiled to lists. By using this functionality, it took just a few steps to produce a working prototype from our specification. Unfortunately, support for this extension had to be dropped when the representation of sets was changed.

Recall that our definition of graph composition involves existential quantification, which is not executable as such. However, graph composition can be made executable by proving the following equations and making them available to the code generator:

$edges\text{-}match\ ((n,\ e,\ m),\ (n',\ e',\ m')) = (m = n')$
$connect\text{-}edges\ ((n,\ e,\ m),\ (n',\ e',\ m')) = (n,\ ee',\ m')$
$(Graph\ G)(Graph\ H) =$
$\quad Graph\ (connect\text{-}edges\ `\ \{x \in G \times H.\ edges\text{-}match\ x\})$

Note that the bounded comprehension and the image operation (`) are executable, as they are compiled to an expression involving *map* and *filter*.

The following function, overloaded on the type class of Kleene algebras, computes transitive closures by a simple iteration:

$$\mathit{mk\text{-}tcl}\ A\ X = (\textsf{if}\ XA \leq X\ \textsf{then}\ X\ \textsf{else}\ \mathit{mk\text{-}tcl}\ A\ (X + XA))$$

Note that *mk-tcl* need not always terminate. However, since the SCT problems we consider are always finite, termination can be proved for these cases. By induction we can prove that *mk-tcl* computes transitive closures of finite graphs:

$$\mathit{finite\text{-}cg}\ A \Longrightarrow \mathit{mk\text{-}tcl}\ A\ A = A^{+}$$

Then the following function checks SCT':

```
$\mathit{test\text{-}SCT}\ \mathcal{A} =$
$(\textsf{let}\ \mathcal{T} = \mathit{mk\text{-}tcl}\ \mathcal{A}\ \mathcal{A}$
 $\textsf{in}\ \forall (n,\ G,\ m) \in \mathit{dest\text{-}graph}\ \mathcal{T}.$
      $n \neq m\ \vee$
      $GG \neq G\ \vee\ (\exists (p,\ e,\ q) \in \mathit{dest\text{-}graph}\ G.\ p = q \wedge e = <))$
where $\mathit{dest\text{-}graph}\ (\mathit{Graph}\ G) = G.$
```

We prove that the function is correct:

Theorem 3.27. $\mathit{finite\text{-}cg}\ \mathcal{C} \Longrightarrow SCT'\ \mathcal{C} = \mathit{test\text{-}SCT}\ \mathcal{C}$

Note that the bounded universal and existential quantifiers in the definition of *test-SCT* do not prevent code generation: They are translated to the corresponding predicates on lists. Hence, *test-SCT* can be translated to ML and executed.

3.10.4 The complete procedure

Connecting the results of the previous subsections, we obtain a method to solve Isabelle termination goals using the unrestricted size-change principle:

- Recast the goal to the form $\mathit{wf}\ (\mathit{mk\text{-}rel}\ [c_1,\ \ldots,\ c_n])$, where $c_1,\ \ldots,\ c_n$ are call descriptors.
- Assign measures to each call, and, following the steps outlined in §3.10.2, construct a size-change problem $\mathcal{C}$.
- Apply Thm. 3.26. It remains to prove $SCT\ \mathcal{C}$.
- Apply Thm. 3.25. By construction, $\mathcal{C}$ is finite, so it remains to show SCT' $\mathcal{C}$.
- Apply Thm. 3.27, obtaining an executable goal.
- Evaluate the goal to *True*, either using the simplifier (which is currently only feasible for very small examples), or by translating to ML first (which requires Isabelle 2007, as pointed out above).

Chapter 4

Pattern Matching

Contents

4.1 Introduction

Pattern matching plays an important role in functional languages, where it is used as a structured and convenient notation for expressing complex branching behaviour. In this chapter we aim to provide support for this style of definition.

The approach discussed in Chapter 2 allows the definition of functions by a single fixed-point equation of the form $f\,x = F\,f\,x$. With pattern matching, our specification now consists of several conditional equations:

$$\begin{aligned}
&\bigwedge vs_1.\ C_1\ vs_1 \Longrightarrow f\ (p_1\ vs_1) = r_1\ f\ vs_1\\
&\quad\vdots\\
&\bigwedge vs_n.\ C_n\ vs_n \Longrightarrow f\ (p_n\ vs_n) = r_n\ f\ vs_n
\end{aligned}$$

In each equation, p_i is the pattern, r_i is the right-hand side, and C_i is a condition. The function f may occur on the right-hand sides as a recursive call, but not in the patterns or conditions.

Figure 4.1 shows various specifications that are instances of this scheme. We will use them as examples in the following discussion.

Pattern matching in functional programming In functional programming, pattern matching is usually restricted to datatypes: Only datatype constructors and variables are allowed in the patterns p_i. This ensures that they can be compiled into efficient tests. Some languages also allow simple invertible arithmetic expressions such as $n + k$, where k is a constant[1]. Haskell and OCaml, but not Standard ML, also support side conditions (called guards). In our schema, the conditions C_i play this role.

The meaning of overlapping patterns is always *sequential*: If more than one pattern matches the input value, then the topmost equation "wins". If the patterns are incomplete, that is, none of them matches the input value, a runtime error occurs. Writing incomplete patterns is sometimes considered bad style, and compilers can issue a warning when detecting them.

Pattern matching in equational specifications The setting of defining HOL functions satisfying the specification schema above is different in two important points, a bad one (1) and a good one (2):

(1) The sequential interpretation of patterns is fundamentally incompatible with the semantics of our specification scheme in HOL, where each equation must hold individually.

(2) As computation is not a concern at definition time, we need not restrict the form of the patterns for the sake of executability and can allow a much wider class of patterns.

The function *sep* in Fig. 4.1(a) illustrates (1): Written with sequential semantics in mind, is is a perfectly valid functional program, which inserts a fixed element between eny two elements of a list. But as a HOL specification, it is inconsistent: The second equation states that the function is a projection on the second argument, which contradicts the first equation. The inconsistency can easily be removed by specializing xs to the instances $[\,]$ and $[x]$ in the second equation. However, we will discuss situations where the fix is much less obvious. Overlapping patterns are not always inconsistent, as the functions gcd and And in Fig. 4.1(d) show: Despite some overlap in the patterns, there is no way to deduce an inconsistency.

Point (2) is much less problematic: Since functions need not be executable, we can drop the syntactic restrictions on the form of the patterns and also allow definitions like in Fig. 4.1(c): Such definitions are perfectly fine as mathematical specifications, but there is no functional language that would admit them.

4.2 General Pattern Matching

In this section we show how to support definitions following the general schema above by encoding them using a special matching combinator. To ensure that the specification is consistent, we generate two proof obligations: a compatibility and a completeness condition.

The proof of compatibility and completeness cannot be automated in general, and must be provided by the user as the fair price to pay for the expressivity

[1]The inclusion of $(n + k)$-patterns in Haskell was highly controversial [55], as it introduces some subtleties in combination with overloading.

$$\begin{aligned} &\textit{sep}\ a\ (x{:}y{:}ys) = x{:}a{:}\textit{sep}\ a\ (y{:}ys)\\ &\textit{sep}\ a\ xs \qquad\quad\ = xs \end{aligned}$$

(a) Inconsistent overlapping specification

$$\begin{array}{lcl} \textit{gcd}\ x\ 0 & = & x\\ \textit{gcd}\ 0\ y & = & y\\ x < y & \Longrightarrow & \textit{gcd}\ (\textit{Suc}\ x)\ (\textit{Suc}\ y) = \textit{gcd}\ (\textit{Suc}\ x)\ (y - x)\\ \neg\ x < y & \Longrightarrow & \textit{gcd}\ (\textit{Suc}\ x)\ (\textit{Suc}\ y) = \textit{gcd}\ (x - y)\ (\textit{Suc}\ y) \end{array}$$

(b) gcd with conditional equations

$$\begin{aligned} &\textit{even}\ (2 * n) \qquad\ = \textit{True}\\ &\textit{even}\ (2 * n + 1) = \textit{False} \end{aligned}$$

(c) Complex patterns

$$\textbf{datatype}\ \textit{three-valued-logic} = T \mid F \mid X$$

$$\begin{aligned} &\textit{And}\ T\ p\ = p\\ &\textit{And}\ p\ T\ = p\\ &\textit{And}\ p\ F\ = F\\ &\textit{And}\ F\ p\ = F\\ &\textit{And}\ X\ X = X \end{aligned}$$

(d) Consistent overlap

Figure 4.1: Examples of definitions with pattern matching

of general pattern matching. However, there is a money-back guarantee, as we must only pay when we really use that extra generality: For disjoint datatype patterns, both conditions can be proved fully automatically.

4.2.1 Compatibility and Completeness

The *compatibility condition* expresses that if a value matches more than one pattern, then all corresponding right-hand sides yield the same function value. In other words, the equations are not contradictory, and the situation in Fig. 4.1(a) cannot occur. For each pair of clauses i and j, we have the following condition:

$$\bigwedge vs_i\ vs_j'.\ p_i\ vs_i = p_j\ vs_j' \Longrightarrow C_i\ vs_i \Longrightarrow C_j\ vs_j' \Longrightarrow r_i\ f\ vs_i = r_j\ f\ vs_j'$$

Note that this condition is trivially satisfied if the patterns are disjoint.

The *completeness condition* expresses that every value matches at least one pattern. It has the form of an elimination rule:

$$\frac{\bigwedge vs_1.\ x = p_1\ vs_1 \Longrightarrow C_1\ vs_1 \Longrightarrow P \qquad \ldots \qquad \bigwedge vs_n.\ x = p_n\ vs_n \Longrightarrow C_n\ vs_n \Longrightarrow P}{P}$$

Although not required for consistency, the completeness condition justifies the case distinction which is part of the induction rule. The elimination rule format is appropriate for the natural deduction approach of Isabelle, but many humans find the following equivalent format more readable:

$$(\exists\, vs_1.\ x = p_1\ vs_1 \wedge C_1\ vs_1) \vee \ldots \vee (\exists\, vs_n.\ x = p_n\ vs_n \wedge C_n\ vs_n)$$

The compatibility condition does not require a specialized method, since it is routinely solved by *auto*, using the distinctness and injectivity properties of datatype constructurs.

We provide a method *pat-completeness*, which proves completeness conditions for datatype patterns by nested case distinctions over the datatypes involved. The TFL package already performs essentially the same proof internally [98, §3.4.2].

Example 4.1. For the function *And* as defined in Fig. 4.1(d) the following completeness condition arises, and is proved automatically:

$$\frac{\bigwedge p.\ x = (T,\ p) \Longrightarrow P \qquad \bigwedge p.\ x = (p,\ T) \Longrightarrow P \qquad \bigwedge p.\ x = (p,\ F) \Longrightarrow P \qquad \bigwedge p.\ x = (F,\ p) \Longrightarrow P \qquad x = (X,\ X) \Longrightarrow P}{P}$$

Note that some of the equations overlap. But compatibility is still easy to prove. Here the condition for the first and the second equation:

$$\bigwedge p\ p'.\ (T,\ p) = (p',\ T) \Longrightarrow p = p'$$

Example 4.2. The function *even* from Fig.4.1(c) produces the following compatibility condition:

$$\frac{\bigwedge n.\ x = 2 * n \Longrightarrow P \qquad \bigwedge n.\ x = 2 * n + 1 \Longrightarrow P}{P}$$

This goal cannot be solved with datatype reasoning. But we can convert it from elimination rule format to the object-logic statement $(\exists\, n.\ x = 2 * n) \vee (\exists\, n.\ x = 2 * n + 1)$ which can be solved automatically since it falls in the realm of presburger arithmetic. Likewise, the compatibility conditions can be fed into a decision procedure.

4.2.2 Implementation using a matching combinator

To support function definitions with general pattern matching, the definition principles described in Chapter 2 can be generalized without major surprises, and the current function package implements this generalization: The inductive definitions of the graph and the domain have n clauses instead of one, and the individual recursive equations are derived from these definitions with the help of the compatibility conditions.

In this section, we take a different route: Instead of generalizing the definition facilities, we will show how a definition with general pattern matching can be reduced to a pattern-free one, which can then be treated as usual. This encoding suggests a two-stage approach, which separates the treatment of recursion and pattern matching, and is thus easier to understand and maintain in the long run. However, the current implementation is still monolithic and handles pattern matching and recursion together. In future work, we hope to make the implementation more modular, based on the following construction.

We define a general matching combinator *MATCH*:

$$\begin{array}{l} \mathit{MATCH} :: (\gamma \Rightarrow \mathit{bool} \times \alpha \times \beta) \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta \\ \mathit{MATCH}\ M\ d\ x = \\ (\textsf{if}\ \exists!r.\ \exists\, v.\ M\ v = (\mathit{True},\ x,\ r) \\ \ \textsf{then}\ \mathit{THE}\ r.\ \exists\, v.\ M\ v = (\mathit{True},\ x,\ r)\ \textsf{else}\ d\ x) \end{array}$$

We call the argument M a *matching clause*, which should be of the form $\lambda v.\ (C\ v,\ p\ v,\ r\ v)$ where C is a condition, p is a pattern, and r is the corresponding result. The *MATCH* combinator takes a value x, and matches it against a matching clause M. The match succeeds iff the result $r\ v$ is unique for all choices of v satisfying $x = p\ v$ and $C\ v$. In this case, the invocation of *MATCH* returns the result. Otherwise, the default value $d\ x$ is returned. By nesting *MATCH* expressions, we can describe sequences of matching clauses.

A matching clause $\lambda v.\ (C\ v,\ p\ v,\ r\ v)$ does not tell us how to effectively compute the result of the match. However, it captures the logical essence of pattern matching.

Using *MATCH*, a definition following our schema can be encoded as a single equation as follows:

$$\dfrac{\begin{array}{l}\bigwedge vs_1.\ C_1\ vs_1 \Longrightarrow (\Gamma_{\mathit{11}} \Longrightarrow P\ r_{\mathit{11}}) \Longrightarrow \ldots \Longrightarrow (\Gamma_{\mathit{1}k_1} \Longrightarrow P\ r_{\mathit{1}k_1}) \\ \hfill \Longrightarrow P\ (p_1\ vs_1) \\ \vdots \\ \bigwedge vs_n.\ C_n\ vs_n \Longrightarrow (\Gamma_{n\mathit{1}} \Longrightarrow P\ r_{n\mathit{1}}) \Longrightarrow \ldots \Longrightarrow (\Gamma_{nk_n} \Longrightarrow P\ r_{nk_n}) \\ \hfill \Longrightarrow P\ (p_n\ vs_n)\end{array}}{P\ a}$$

Figure 4.2: Induction rule for specifications with pattern matching

$$\begin{array}{l} f\ x = \\ \quad \mathit{MATCH}\ (\lambda(vs_1).\ (C_1\ vs_1,\ p_1\ vs_1,\ r_1\ f\ vs_1)) \\ \quad (\mathit{MATCH}\ \ldots \\ \qquad \vdots \\ \qquad (\mathit{MATCH}\ (\lambda(vs_n).\ (C_n\ vs_n,\ p_n\ vs_n,\ r_n\ f\ vs_n)) \\ \qquad\ (\lambda x.\ \mathit{undefined}))\ldots)\ x \end{array}$$

Here, $(\lambda(vs_i).\ \ldots)$ denotes an abstraction over a tuple of variables. Recall that *undefined* is an unspecified constant.

Example 4.3. The *gcd* function from Fig. 4.1(b) can be written using the *MATCH* combinator as follows:

$$\begin{array}{l} \mathit{gcd}\ x\ y = \\ \mathit{MATCH}\ (\lambda x.\ (\mathit{True},\ (x,\ \mathit{0}),\ x)) \\ \ (\mathit{MATCH}\ (\lambda y.\ (\mathit{True},\ (\mathit{0},\ y),\ y)) \\ \quad (\mathit{MATCH}\ (\lambda(x,\ y).\ (x < y,\ (\mathit{Suc}\ x,\ \mathit{Suc}\ y),\ \mathit{gcd}\ (\mathit{Suc}\ x)\ (y - x))) \\ \qquad (\mathit{MATCH} \\ \qquad\quad (\lambda(x,\ y).\ (\neg\ x < y,\ (\mathit{Suc}\ x,\ \mathit{Suc}\ y),\ \mathit{gcd}\ (x - y)\ (\mathit{Suc}\ y))) \\ \qquad\quad (\lambda x.\ \mathit{undefined})))) \\ \ (x,\ y) \end{array}$$

This definition can be processed by the function package in the normal way, provided that a suitable congruence rule for *MATCH* is present. It remains to derive the original specification from this definition.

Proving the original equations We now show how the ith conditional equation

$$\bigwedge vs_i.\ C_i\ vs_i \Longrightarrow f\ (p_i\ vs_i) = r_i\ f\ vs_i$$

can be derived from the pattern-free definition with the help of the compatibility condition.

Proof sketch. We unfold the definition of f in the goal.

<u>Case $i = 1$:</u> We must show that the first match succeeds, i.e., $\exists!r.\ \exists\, us.\ C_1\ us \wedge p_1\ us = p_1\ vs_1 \wedge r_1\ us = r$. Existence is trivial with $r = r_1\ vs_1$ and $us = vs_1$. Uniqueness is a direct consequence of the compatibility condition for $i = j = 1$.

<u>Case $i > 1$:</u> This case is simple if the match fails, because we can the remove the outermost use of *MATCH* and proceed in the same way with the following matching clauses until we encounter the right one. However, the match may

datatype $color = R \mid B$
datatype $\alpha\ rbt = E \mid T\ color\ (\alpha\ rbt)\ \alpha\ (\alpha\ rbt)$

$balance :: \alpha\ rbt \Rightarrow \alpha\ rbt$

$$
\begin{array}{ll}
balance\ (T\ B\ (T\ R\ (T\ R\ a\ x\ b)\ y\ c)\ z\ d) & = T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d) \\
balance\ (T\ B\ (T\ R\ a\ x\ (T\ R\ b\ y\ c))\ z\ d) & = T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d) \\
balance\ (T\ B\ a\ x\ (T\ R\ (T\ R\ b\ y\ c)\ z\ d)) & = T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d) \\
balance\ (T\ B\ a\ x\ (T\ R\ b\ y\ (T\ R\ c\ z\ d))) & = T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d) \\
balance\ t & = t
\end{array}
$$

Figure 4.3: Pattern matching in the *balance* function

also succeed if the ith clause overlaps with the first. However, in that case, the compatibility condition implies that the result of the match will nevertheless be equal to $r_i\ vs_i$.

Induction rule for pattern matching The structure of the pattern matching specification is also reflected in the induction rule, which has one case for each equation. This built-in case distinction leads to proofs that have the same structure.

Figure 4.2 shows the total induction rule, where $[\Gamma_{i1} \rightsquigarrow r_{i1}], \ldots, [\Gamma_{ik_i} \rightsquigarrow r_{ik_i}]$ are the recursive calls extracted from the ith equation. The right-hand sides of the equations, which were also denoted by r above, do not appear in the induction rule.

4.3 Pattern Disambiguation and Minimization

We have already demonstrated the problems with sequential pattern matching on the function *sep* in Fig. 4.1(a). To obtain a consistent HOL specification, we must first disambiguate the equations by removing overlap between the different clauses, thus recovering the set of equations that was actually meant. In other words, we are considering the transformation of a functional program with overlapping patterns into a term rewrite system, where rules can be applied in any order.

Slind's TFL package implements this disambiguation as part of the compilation of pattern matching case expressions. As the leaves of the case expression correspond to mutually disjoint patterns, an unambiguous specification can be generated from the case expression.

However, this approach has some weaknesses: First, the size of the resulting specification is not minimal. Second, the form and size of the result can depend on irrelevant details like the order of the function's arguments. This is because the algorithm internally needs to choose a variable on which to do a case split, and it always chooses the first variable. Sometimes, choosing a different variable would lead to a simpler set of equations.[2]

Fig. 4.3 shows a spectacular instance of the problem. The *balance* function (due to Okasaki [83]) implements an operation used for rebalancing red-black

[2]Lucas Dixon implemented a postprocessing operation that mitigates this effect by merging equations again that have been split too much. However, even then the results are not always minimal.

trees. Its heavy use of pattern matching is part of the reason why it is much more elegant than implementations in an imperative language. Here, a complex series of primitive tests is expressed in just four equations that look symmetric, plus a default case.

Unfortunately, all the elegance is suddenly gone when we make the specification disjoint by instantiating it in the same way as we have seen for *is-empty*. If we split up using Slind's algorithm, we end up with a total of 91 (!) equations. Although many of the patterns that arise correspond to ill-formed trees that violate the red-black tree invariant, they must be generated at that point, since they belong to the specification of the balance function, which is defined on the whole free datatype *rbt*. In general, this disambiguation can lead to an exponential blowup.

In this section, we study how to perform this transformation in a way that the number of resulting equations is minimal. This is particularly important, as the size of the specification critically influences the size and complexity of subsequent proofs: e.g., induction proofs about a recursive function usually split up into as many cases as there are defining equations.

After pinpointing the underlying problem, we exhibit a nice correspondence to the well-known problem of minimizing boolean formulas. This link allows us to prove that we are indeed dealing with a hard optimization problem (it is Σ_2^P-complete). Then we describe a method for finding minimal patterns, which is inspired by known results on boolean minimization and generalizes the classical Quine-McCluskey algorithm [71]. We implemented a prototype of the algorithm in Haskell. Despite the discouraging complexity results, its performance is acceptable on the problem sizes we encounter in practice, but in some instances the blowup in the number of equations is unavoidable.

4.3.1 Notation and Problem definition

We must introduce some notation to be able to describe the pattern minimization problem formally. Since function types and polymorphism are not relevant for pattern matching, we can pretend that we live in a monomorphic first-order language and work with many-sorted first-order terms for a fixed set of sorts $\mathcal{S}$ and a finite sorted signature Σ. Note that this notion of sorts has nothing to do with the sorts used in Isabelle's type-class mechanism. Sort are most similar to types, but they have no structure.

For pattern matching, only constructor terms are relevant, so we assume that all function symbols in Σ are constructors. Then the sort-indexed family of terms $(\mathcal{T}_s)_{s\in\mathcal{S}}$ is defined inductively as usual. A term is called a *linear* term or a *pattern* if no variable occurs more than once. $\mathcal{P}_s$ denotes the set of patterns of sort s and $\mathcal{G}_s$ denotes the set of ground terms (i.e., terms without variables). We write Σ_s for the sets of constructors for values of sort s.

Since patterns must be linear and we are only concerned with matching, we can replace all variables by the wildcard symbol $*$. Formally, the wildcard carries a sort annotation ($*_s$), such that the sort of a term is always uniquely defined. However, we liberally drop sort annotations that are clear from the context.

We write $s \preceq t$ if s is an instance of t (and $s \prec t$ if it is a proper instance, i.e., $s \neq t$). Given a pattern p, we can express its semantics as the set of all ground instances:

$$[p] := \{g \in \mathcal{G} \mid g \preceq p\}$$

For finite sets of patterns we set $[P] := \bigcup_{p \in P}[p]$.

Computing the intersection $p \wedge q$ of two patterns is a degenerate case of unification. Intersection is a partial operation: if the patterns are disjoint, we write $p \wedge q = \bot$, although $\bot$ itself is not a pattern. We write $\sup(p, q)$ for the supremum with respect to $\preceq$: E.g. $\sup(f(a, *), f(*, b)) = f(*, *)$. More generally, $(\mathcal{P}_s \cup \{\bot\}, \preceq)$ is a complete lattice, a special case of the subsumption lattice for terms described by Huet [56].

For our examples and constructions, we implicitly assume that Σ contains the constructors T and F for booleans, $\mathsf{0}$ and Suc for naturals and suitable constructors for n-tuples, written $\langle \cdot, \cdots, \cdot \rangle$. We ignore currying for the presentation and assume that function arguments are always tupled. For example, the patterns of the function in Fig. 4.1(a) are written $\{\langle *, \mathsf{Cons}(*, \mathsf{Cons}(*, *))\rangle, \langle *, * \rangle\}$.

Observe that we may have $[p] = [q]$ and $p \neq q$: For example, $[C(*, \ldots, *)] = [*]$ if C is the only constructor of the given sort. To resolve this ambiguity, we define $\lceil p \rceil = \sup \{q \in \mathcal{P} \mid [q] = [p]\}$. Note that $\lceil p \rceil$ can easily be computed from p.

Pattern minimization and complement

We can now state the problems formally and relate them to the informal discussion above. As usual, optimization problems are stated as decision problems:

> Disambiguation: Given patterns $p_1, \ldots, p_n$ and an integer k, are there sets of patterns $P_1, \ldots, P_n$, such that for each $i \in \{1, \ldots, n\}$, $[P_i] = [p_i] \setminus \bigcup_{j=1}^{i-1}[p_j]$, and the number of patterns in $P_1 \cup \ldots \cup P_n$ is less than k?

Note that we do not require that *all* the resulting patterns be non-overlapping. Within one group P_i, the patterns may overlap, since they are stemming from the same equation (with the same right-hand side).

To approach this problem, we study some related problems which are slightly simpler to express, like that of building a complement:

> Pat Complement: Given a finite set P of patterns and an integer k, is there a set Q of at most k patterns, such that $[Q] = \mathcal{G} \setminus [P]$?

The related problem of just minimizing a set of patterns is given as follows:

> Min Pat: Given a finite set P of patterns and an integer k, is there a set P' of at most k patterns, such that $[P'] = [P]$?

Technical note: For the complement-like problems, the bound k must be given in unary notation. This avoids that we merely measure the output complexity. A set of patterns can grow exponentially under complementation, so any algorithm computing it must take exponential time. If k is in unary, then the size of the complemented patterns is again polynomial in the size of the input, and we are measuring the complexity of the actual optimization process, not its result. The same technique is used by Umans [106].

Example 4.4. With the following patterns over a type with three nullary constructors A, B, and C, disambiguation must lead to an exponential blowup:

$$\begin{array}{l}\langle A,*,*,*,*\rangle\\ \langle *,A,*,*,*\rangle\\ \langle *,*,A,*,*\rangle\\ \langle *,*,*,A,*\rangle\\ \langle *,*,*,*,A\rangle\\ \langle *,*,*,*,*\rangle\end{array}$$

It is easy to see that the last pattern stands for all value combinations that do not contain A. But this set of values cannot be expressed compactly by some patterns, since any pattern that has a wildcard at some position cannot be a candidate for the last equation because it would match a term having an A at that position. Thus, the patterns for the default case cannot have any wildcards, and therefore we need all combinations of B and C.

While the example above demonstrates a blowup that is unavoidable, the following contrived example shows that, in theory, an optimization can sometimes save us from an exponential blowup:

Example 4.5. For a given $n \in \mathbb{N}$, we construct a function with $\frac{n(n-1)}{2}$ boolean arguments, and with n equations. We assign indices $1, \ldots, n$ to the equations and associate a pair (i, j) with $1 \leqslant i < j \leqslant n$ to each argument position. Now equation k has at argument position (i, j) the pattern T, if $k = i$ and F if $k = j$. Otherwise there is a wildcard pattern $*$.

For $n = 3$, this construction yields a variation of the diagonal function (cf. Wadler [110]):

$$\begin{array}{l}\mathit{diagonal} :: \mathit{Bool} \to \mathit{Bool} \to \mathit{Bool} \to \mathit{Int}\\[4pt] \mathit{diagonal}\ T\ T\ \_ = 1\\ \mathit{diagonal}\ F\ \_\ T = 2\\ \mathit{diagonal}\ \_\ F\ F = 3\end{array}$$

Since the equations i and j have different patterns at argument position (i, j), they are all disjoint. Hence the optimal disambiguation is to leave everything as it is. However, the naive disambiguation fails to recognize this and produces exponentially many equations.

Now let us look at the relationship between the different problems. Clearly Pat Complement cannot be any harder than Disambiguation, as we can see from the *balance* example: A catch-all pattern in the end will be replaced by the complement of the preceding patterns.

The following lemma shows that the two problems are actually equivalent:

Lemma 4.6. Disambiguation *can be reduced* (*in P-time*) *to* Pat Complement.

Proof. We do a reduction between the optimization problems, showing how to disambiguate optimally if we can compute minimal complements.

We first show how we can use Pat Complement to *subtract* a set of patterns Q from a pattern p:

First compute $Q' = \{p \wedge q \mid q \in Q,\ p \wedge q \neq \bot\}$. Now consider all the positions where p has a wildcard and call them $\pi_1, \ldots, \pi_n$. Since the patterns in Q' are instances of p, they can only differ at these positions. We remove the outer structure and replace it by a tuple: $Q'' = \{\langle q|_{\pi_1}, \ldots, q|_{\pi_n}\rangle \mid q \in Q'\}$. We can now solve Pat Complement for Q'' to compute the minimal pattern set C with $[C] = \mathcal{G} \setminus Q''$. We obtain the result of the subtraction by adding the term structure of p again: $R = \{p[c_1, \ldots, c_n] \mid \langle c_1, \ldots, c_n\rangle \in C\}$. Since C is minimal, R must also be minimal.

Disambiguation is now easily reduced to multiple subtractions. $\square$

Let us demonstrate this reduction by a small example: Consider the datatype
datatype $T = \mathsf{A} \mid \mathsf{B}\ \mathit{nat} \mid \mathsf{C}\ (\mathit{nat} \times \mathit{nat})$
and suppose we want to compute

$$\begin{array}{rl} & \langle \mathsf{C}(*,*), \quad * \quad\rangle \\ - & \langle \mathsf{C}(0,0), \quad \mathsf{A} \quad\rangle \\ - & \langle \quad * \quad, \mathsf{B}(\mathsf{Suc}(*))\rangle \end{array}$$

To subtract the third and second from the first pattern, we first compute the intersections, obtaining

$$\begin{array}{rl} & \langle \mathsf{C}(*,*), \quad * \quad\rangle \\ - & \langle \mathsf{C}(0,0), \quad \mathsf{A} \quad\rangle \\ - & \langle \mathsf{C}(*,*), \mathsf{B}(\mathsf{Suc}(*))\rangle \end{array}$$

We remove the outer term structure that is common to all terms and replace it by a tuple. The first pattern is now a universal pattern, hence we have reduced the problem to computing a complement:

$$\begin{array}{rcccccccl} & \langle & * & , & * & , & * & \rangle \\ - & \langle & 0 & , & 0 & , & \mathsf{A} & \rangle \\ - & \langle & * & , & * & , & \mathsf{B}(\mathsf{Suc}(*)) & \rangle \\ \hline = & \langle & \mathsf{Suc}(*) & , & * & , & \mathsf{A} & \rangle \\ & \langle & * & , & \mathsf{Suc}(*) & , & \mathsf{A} & \rangle \\ & \langle & 0 & , & 0 & , & \mathsf{B}(0) & \rangle \\ & \langle & * & , & * & , & \mathsf{C}(*,*) & \rangle \end{array}$$

After that, we just add the outer structure $\langle \mathsf{C}(\cdot,\cdot), \cdot\rangle$ again, and obtain the result of the subtraction.

4.3.2 Complexity Results

We now show that our pattern minimization problems can encode the well known-problem of minimizing boolean formulas in Disjunctive Normal Form (DNF).

This problem has already recieved a lot of attention, as it is crucial for the design of digital circuits. Many exact and heuristic methods have been studied, the most well-known probably being the classical algorithm by Quine & McCluskey [71], on which we will base our pattern minimization algorithm in §4.3.3.

Despite the high practical importance, the exact complexity of the problem has only recently been settled, when Umans proved it Σ_2^P-complete in his PhD thesis [107]. The complexity class Σ_2^P belongs to the polynomial hierarchy and contains the problems that can be solved by a nondeterministic Turing machine with access to a SAT oracle that it can use to solve NP-complete problems in a single step[3]. Σ_2^P can be seen as "one level up" from NP, and its canonical complete problem is QSAT_2, the satisfiability problem for formulas of the form $\exists\vec{x}\,\forall\vec{y}.\,\phi(\vec{x},\vec{y})$ where $\vec{x}$ and $\vec{y}$ are vectors of boolean-valued variables. (For more details, see Papadimitriou [87].)

The DNF minimization problems can be stated as follows:

> MINIMUM EQUIVALENT DNF (MIN DNF): Given a DNF formula ϕ and an integer k, is there a DNF formula equivalent to ϕ with at most k terms[4]?
>
> SHORT CNF: Given a formula ϕ in Conjunctive Normal Form (CNF) and an integer k in unary notation, is there a DNF formula equivalent to ϕ with at most k terms?

Both problems are known to be complete for Σ_2^P [94].

The central idea in showing that MIN PAT is Σ_2^P-complete is that DNF formulas can be mapped to patterns:

Definition 4.7. *Let ϕ be a boolean DNF formula with variables v_1 through v_n. It has the form $\phi = t_1 \vee \cdots \vee t_k$, where each t_i is a conjunction of literals, which we view as a set. Then*

$$E(\phi) = \{\ \langle p_1^1, \ldots, p_1^n\rangle\ , \ldots ,\ \langle p_k^1, \ldots, p_k^n\rangle\ \}$$

where

$$p_i^j = \begin{cases} \mathsf{T} & \text{if } v_i \in \mathit{Literals}(t_j) \\ \mathsf{F} & \text{if } \neg v_i \in \mathit{Literals}(t_j) \\ * & \text{otherwise} \end{cases}$$

For example $E(v_1\bar{v}_3 \vee \bar{v}_2 v_3) = \{\langle \mathsf{T}, *, \mathsf{F}\rangle, \langle *, \mathsf{F}, \mathsf{T}\rangle\}$.
Clearly, $\phi(b_1, \ldots, b_n)$ is true iff $\langle b_1, \ldots, b_n\rangle \in [E(\phi)]$.

Theorem 4.8 (Lower Bounds)**.**

1. *Deciding whether a given set of patterns P is incomplete (i.e. $[P] \neq [*]$) is NP-hard.*
2. MIN PAT *is Σ_2^P-hard.*
3. PAT COMPLEMENT *is Σ_2^P-hard.*

Proof. We reduce from the related boolean problems using the embedding from Definition 4.7.

1. Reduction from SAT: Let ϕ be in CNF. Using De Morgan's laws we produce the DNF formula ψ equivalent to $\neg\phi$. Then

$$[E(\psi)] \neq [*] \quad \text{iff} \quad \psi \text{ is not a tautology} \quad \text{iff} \quad \phi \text{ is satisfiable.}$$

[3]This is more powerful than the "guessing" facility of nondeterminism, since it can also detect when no solution exists.

[4]In the terminology of boolean minimization, the word *term* specifically means a disjunct in a DNF. They should not be confused with the first-order terms that we use as patterns or even Isabelle terms.

2. Reduction from MIN DNF.
3. Reduction from SHORT CNF, again interpreting the CNF formula as a negated DNF formula.

□

The incompleteness problem is interesting, since it is actually solved by most compilers of functional languages, which can issue a warning when the patterns of a function definition do not cover all cases. However, the exponential behaviour of the implementations does not seem to pose any difficulties in practice, since the problem instances are usually small.

By a simple guess-and-check argument, we can show that MIN PAT and PAT COMPLEMENT are also contained in Σ_2^P:

Lemma 4.9. *The equivalence problem of two pattern sets is in co-NP.*

Proof. For given sets of patterns P and P', we can nondeterministically choose a ground term, and check whether it is either covered by both P and P', or by none of them. □

Theorem 4.10. MIN PAT *and* PAT COMPLEMENT *are* Σ_2^P*-complete.*

Proof. To show that MIN PAT $\in \Sigma_2^P$, note that a nondeterministic Turing machine with access to a SAT oracle can solve our problem as follows: For a given input P and integer k, it nondeterministically guesses a pattern set P' of size k. It remains to check if $[P] = [P']$. Due to Lemma 4.9, this can be done by the SAT oracle.

For PAT COMPLEMENT $\in \Sigma_2^P$, a similar argument works. Then, with Thm. 4.8 we have completeness for both problems. □

4.3.3 A Minimization Algorithm

Exploiting the similarity to boolean mimization, we can develop an algorithm that computes minimal patterns. The algorithm is a generalization of the well-known Quine-McCluskey method [71]. We focus on the MIN PAT problem first, but with simple modifications, we can also use the procedure to solve PAT COMPLEMENT and DISAMBIGUATION.

The Quine-McCluskey algorithm proceeds as follows to minimize a formula ϕ:

1. Write ϕ in canonical disjunctive normal form, i.e., as a disjunction of "minterms". These are products (i.e. conjunctions) of literals where each variable occurs either positively or negatively. Minterms correspond to the entries in the truth table where ϕ becomes true.
2. From the minterms, constuct the "most general terms that imply ϕ", that is, conjunctions of literals that imply ϕ, but when one literal is removed, the result does not imply ϕ. These terms are called *prime implicants*.
3. Find a minimal subset of prime implicants that covers all minterms of ϕ. Then the minimized formula is the disjunction of these prime implicants.

Example 4.11.

1. Consider the following formula in canonical disjunctive normal form:

$$\phi = \bar{x}\bar{y}\bar{z}\bar{w} \vee \bar{x}y\bar{z}\bar{w} \vee x\bar{y}\bar{z}\bar{w} \vee \bar{x}y\bar{z}w \vee \bar{x}yz\bar{w} \vee$$
$$x\bar{y}\bar{z}w \vee x\bar{y}z\bar{w} \vee \bar{x}yzw \vee \bar{x}\bar{y}z\bar{w} \vee xyzw$$

   The terms of the disjunction are the (positive) minterms. They correspond to entries in the truth table for ϕ.

2. The prime implicants are those terms that cannot be generalized further without leaving ϕ:

$$\{\bar{x}\bar{z}\bar{w}, \bar{y}\bar{z}\bar{w}, x\bar{y}\bar{z}, x\bar{y}\bar{w}, x\bar{z}w, \bar{x}y, yw\}$$

3. By choosing a minimal set of prime implicants that cover all the minterms, we obtain a minimized formula:

$$\phi = x\bar{y}\bar{w} \vee x\bar{z}w \vee \bar{x}y$$

Step 1, which is often implicit in textbook descriptions [59, 72], means that we basically start from the full truth table of the function. Note that the number of minterms is typically exponential in the size of ϕ.

The exact method of combining the minterms to prime implicants in Step 2 is often only vaguely described in textbooks, and if it is described, the algorithm often takes exponential time. However, Strzemecki showed [101] that this step can be done in polynomial time.

Finally, it remains to solve a covering problem in Step 3, which is known to be NP-hard (even in the particular instances arising here [106]).

In the following, we will see that this algorithm can be extended to the more general problem on patterns.

We adapt some terminology from boolean minimization: For a fixed pattern-set P and a pattern p, we say that p is an *implicant* iff $[p] \subseteq [P]$. An implicant is called *prime* if none of its proper generalizations is an implicant.

Obviously, a minimal covering can be constructed from the prime implicants: Any other patterns in a minimal covering could simply be generalized to some prime implicant.

Minterms

Our more general setting is different in one important aspect: In the boolean case, the base set we are considering is just the finite product space $\{0,1\}^n$, whereas the set underlying our patterns is a possibly infinite set of terms. So it is not immediately clear what corresponds to the "truth table" of a boolean function.

However, the nature of pattern matching is still finitary in a certain sense, which allows us to generalize the boolean methods. The idea is to define inductively a set of terms, depending on the patterns we want to minimize, which behaves similarly to the product space.

These terms, called *minterms*, are mutually non-overlapping and cover all of $\mathcal{G}$. Furthermore, they respect the structure of P, in the sense that for a minterm m and a pattern $p \in P$ either $m \preceq p$ or $m \wedge p = \bot$.

Definition 4.12 (Projection). *For $P \subseteq \mathcal{P}_s$, $C \in \Sigma_s$ and $i \leqslant arity(C)$, we define the* projection

$$\Pi_{C,i}(P) = \begin{cases} \{p_i \mid C(p_1, \dots, p_n) \in P\} \cup \{*\} & \text{if } * \in P \\ \{p_i \mid C(p_1, \dots, p_n) \in P\} & \text{otherwise} \end{cases}$$

For example, $\Pi_{\langle\rangle,2}(\{\langle *, \mathsf{Suc}(*)\rangle, \langle 0, *\rangle\}) = \{\mathsf{Suc}(*), *\}$, and $\Pi_{\mathsf{Suc},1}(\{0, \mathsf{Suc}(0), \mathsf{Suc}(*)\}) = \{0, *\}$.

Definition 4.13 (Minterms). *We compute the set of minterms $MT(P)$ recursively as follows:*

$$MT(P) = \begin{cases} \{*\} & \text{if } P = \emptyset \text{ or } P = \{*\} \\ \bigcup_{C \in \Sigma_s} C(MT(\Pi_{C,1}(P)), \dots, MT(\Pi_{C,n}(P))) & \\ & \text{otherwise} \end{cases}$$

Note that above the constructor C is lifted to sets:

$$C(A_1, \dots, A_n) = \{C(a_1, \dots, a_n) \mid a_1 \in A_1 \dots a_n \in A_n\}$$

For example, $MT(0) = \{0\} \cup \mathsf{Suc}(MT(\Pi_{\mathsf{Suc},1}(\{0\}))) = \{0\} \cup \mathsf{Suc}(MT(\emptyset)) = \{0, \mathsf{Suc}(*)\}$.

We divide the set of minterms into *positive* ones that lie within $[P]$ and *negative* ones that are outside:

$$\begin{aligned} M_P^+ &= \{m \in MT(P) \mid \exists p \in P.\ m \preceq p\} \\ M_P^- &= MT(P) \setminus M_P^+ \end{aligned}$$

Example 4.14. For $P = \{\langle \mathsf{Suc}(0), *\rangle, \langle *, 0\rangle\}$, we have

$$\begin{aligned} M_P^+ = \{&\langle \mathsf{Suc}(0), 0\rangle, && \langle \mathsf{Suc}(0), \mathsf{Suc}(*)\rangle, \\ &\langle 0, 0\rangle, && \langle \mathsf{Suc}(\mathsf{Suc}(*)), 0\rangle\} \\ M_P^- = \{&\langle 0, \mathsf{Suc}(*)\rangle, && \langle \mathsf{Suc}(\mathsf{Suc}(*)), \mathsf{Suc}(*)\rangle\}\,. \end{aligned}$$

Defined like this, minterms satisfy the following properties:

Lemma 4.15 (Properties of minterms). *Let $p \in P$ and $m \in MT(P)$.*

(a) The elements of $MT(P)$ are pairwise disjoint, and $[MT(P)] = \mathcal{G}$.

(b) If $m \wedge p \neq \bot$ then $m \preceq p$

(c) $[m] \subseteq [P]$ iff $\exists p \in P. m \preceq p$

(d) For any pattern set $A \subseteq \mathcal{P}_s$, we have

$$[A] \subseteq [P] \iff \forall m \in M_P^-.\ \forall a \in A.\ m \wedge a = \bot$$

Proof.

(a) Simple induction

(b) By induction on p. For $p = *$ the statement is trivial. If $p = C(p_1, \ldots, p_n)$, then $* \notin MT(P)$ and thus m must also start with a constructor. Since $m \wedge p \neq \bot$, we know that $m = C(m_1, \ldots, m_n)$ with $m_i \in MT(\Pi_{C,i}(P))$ and $m_i \wedge p_i \neq \bot$. By induction hypothesis we get $m_i \preceq p_i$ for each i, and thus $m \preceq p$.

(c) For the forward implication, assume $[m] \subseteq [P]$. Since $[m]$ cannot be empty, m must overlap with at least one element of P, and we can apply (b). The reverse implication is immediate.

(d) Obvious, since $[M_P^-] = \mathcal{G} \setminus [P]$.

□

Hence we can see M_P as a partitioning of $\mathcal{G}$, where each partition is represented by a minterm. Moreover, the partitioning is fine enough to be compatible with the shape of $[P]$.

Note that Lemma 4.15(d) enables us to check algorithmically if a given pattern (or set of patterns) is an implicant. This check will be required in the next sections.

Constructing Prime Implicants

The next step in the Quine-McCluskey procedure requires finding the prime implicants of the boolean formula.

In textbooks this is often done by repeatedly joining minterms to larger terms, until a fixpoint is reached. However, this procedure can have exponential runtime, which is unnecessary.

Strzemecki describes how to obtain prime implicants in polynomial time [101]. Unfortunately, the paper is a little hard to read due to a lot of nonstandard and redundant notation. However, it can be reduced to a few simple ideas, which generalize nicely to our pattern world.

Lemma 4.16. *Let p be an implicant for P and $m \in M_P^+$ a minterm such that $m \preceq p$. Then there exists a minterm $m' \in M_P^+$, such that $[p] = [\sup(m, m')]$.*

Proof. We proceed by induction on m.

If $m = *$, choose $m' = *$.

Assume $m = C(m_1, \ldots, m_n)$. If $p = *$, we simply choose any $m' \in M_P^+$ whose topmost constructor is different from C. If this doesn't exist, then C must be the only constructor of the respective sort and we know that $[p] = [C(*, \ldots, *)]$ and we proceed as if p had that form.

If $p = C(p_1, \ldots, p_n)$, we have $m_i \prec p_i$, and the m_i are minterms. Applying the induction hypothesis we obtain minterms $m_i' \in M_{\Pi_{C,i}(P)}$ such that $p_i = \sup(m_i, m_i')$. We get $m' = C(m_1', \ldots, m_n') \in M_P$. And clearly $\sup(m, m') = p$. □

Corollary 4.17. *Every prime implicant can be written as $\lceil \sup(m, m') \rceil$ with some $m, m' \in M_P^+$*

This implies a simple polynomial-time algorithm for finding all prime implicants: Build the suprema of all possible pairs of positive minterms and remove those patterns that are not implicants (using Lemma 4.15(d)) or instances of others.

Essential Prime Implicants

A prime implicant is called *essential* if it covers a minterm not covered by any other prime implicant. Since essential prime implicants must necessarily appear in the minimal covering, it is a useful optimization to generate them first. After this, only the remaining minterms must be covered by other prime implicants.

Also here, we can generalize Strzemecki's work, redefining the relevant notions for our framework:

Definition 4.18. *Informally, the set $G(t)$ of* simple generalizations *of t is the set of terms obtained from t by replacing exactly one non-wildcard subterm by $*$.*

Formally, $G(t)$ is defined recursively as follows:

$$\begin{aligned} G(*) &= \{\} \\ G(C(t_1, \ldots, t_n)) &= \{*\} \cup \bigcup_{i=1}^{n} C(t_1, \ldots, G(t_i), \ldots, t_n) \end{aligned}$$

Lemma 4.19. *If $s \prec t$ then t is the supremum of some subset of $G(s)$.*

Proof. Informally, for each missing constructor in t compared to s, we include the corresponding element of $G(s)$ in the subset. Formally, use induction. □

Definition 4.20 (Neighbourhood terms)**.** *For given P and $m \in M_P^+$, we define the* neighbourhood term *of m by $R_P(m) = \sup \{g \in G(m) \mid [g] \subseteq [P]\}$.*

Lemma 4.21. *Let $m \in M_P^+$ and $r = R_P(m)$. If $[r] \subseteq [P]$, then r is an essential prime implicant.*

Proof. Assume that $r = R_P(m)$ is an implicant. In order to see that r is prime, we show that every implicant i which contains m must be subsumed by r. So fix i with $m \preceq i$ and $[i] \subseteq [P]$. If $i = m$, then obviously $i \preceq r$. Otherwise we have $m \prec i$, and Lemma 4.19 yields $i = \sup G$ for some $G \subseteq G(m)$. Since i is an implicant, each element of G must also be, and thus $i \preceq R_P(m) = r$. Since i was arbitrary, we know that any prime implicant containing m must be equal to r, and thus r is essential to cover m. □

In fact, the converse also holds and all essential prime implicants have the form above:

Lemma 4.22. *Every essential prime implicant equals $R_P(m)$ for some $m \in M_P^+$.*

Proof. Let e be an essential prime implicant. Then there exists an $m \preceq e, m \in M_P^+$ covered by no other prime implicant. Then e is an upper bound for $\{g \in G(m) \mid [g] \subseteq [P]\}$, which implies $R_P(m) \preceq e$. Since e is an implicant, we have $[R_P(m)] \subseteq [P]$. Applying the above lemma, we get the reverse inequality $e \preceq R_P(m)$, and thus $e = R_P(m)$. □

So, to compute all essential prime implicants, it suffices to compute $R_P(m)$ for every $m \in M_P^+$, and filter out those that are not implicants.

Overall Algorithm

Given a pattern set P to minimize, we proceed as follows:

1. Compute M_P^+ and M_P^-.
2. Compute the essential prime implicants E, as described in §4.3.3, and determine the set $\bar{M} = \{m \in M_P^+ \mid [m] \nsubseteq [E]\}$.
3. Compute the set N of nonessential prime implicants (containing minterms from $\bar{M}$) as described in §4.3.3.
4. Find a covering of $\bar{M}$ by a subset N' of N.
5. Return $E \cup N'$.

Note that if we want to minimize the complement instead, we can just swap the roles of M_P^+ and M_P^-. For Disambiguation, we just partition the minterms into n classes, one for each of the original equations, and perform the other steps for each of the classes accordingly.

4.3.4 Implementation and Experiments

We implemented a prototype of the procedure in Haskell and tested it on a small suite of examples stemming from user-contributed theories to the library of the Isabelle/HOL prover. From 232 definitions, we filtered out those that just use trivial pattern matching on one of the arguments, without any nesting. The 97 remaining examples were minimized by our prototype in less than 3 seconds on a 1.2 GHz laptop, and no individual example took more than half a second to process. In 16% of the cases, an improvement over the disambiguation method implemented in TFL could be made.

This indicates that minimization is feasible and occasionally useful for definitions occurring in theorem proving practice. However, such figures always require a good amount of scepticism, as the sample is influenced by the restrictions of previous Isabelle versions. It reflects what people *could* already do, not what they *would like to do*. In particular, some functions in the developments of arithmetic decision procedures [27, 28] were developed in a way that tries to avoid excessive blowup.

In the following, we briefly present three interesting examples:

Balance Our initial example, the balancing function for red-black trees, has five equations, which split up to 91 with TFL. Our minimization algorithm computes a minimum number of 59 patterns.

Interpreter Function The function *interp* in Fig. 4.4 arises from a crude interpreter for a simplistic expression language. Operator names are modeled as natural numbers (and here the number literals just abbreviate terms built from 0 and *Suc*). Values are either booleans, numerical values or undefined. We omitted the right-hand sides since they are not relevant for the matching.

This specific example was given to the author by Tobias Nipkow, complaining that Isabelle produced too many equations, when disambiguating the definition. TFL produced either 36 or 39 cases, depending on the order of function

```
datatype val = Nv nat | Bv bool | Undef

interp :: nat ⇒ val list ⇒ val

interp 0 []                = ...
interp 1 [Nv n]            = ...
interp 2 [Nv m, Nv n]      = ...
interp 3 []                = ...
interp 4 [Bv b]            = ...
interp 5 [Bv b1, Bv b2]    = ...
interp 6 [Nv n1, Nv n2]    = ...
interp k xs                = ...
```

Figure 4.4: Interpreter Function

```
datatype T = C nat | Bound nat | Neg T | Add T T | Sub T T | Mul nat T

numadd :: T ⇒ T ⇒ T

numadd (Add (Mul c1 (Bound n1)) r1) (Add (Mul c2 (Bound n2)) r2) = ...
numadd (Add (Mul c1 (Bound n1)) r1) t                             = ...
numadd t (Add (Mul c2 (Bound n2)) r2)                             = ...
numadd (C b1) (C b2)                                              = ...
numadd a b                                                        = ...
```

Figure 4.5: Numadd

arguments. By manual inspection, we were able to express the function in just 31 equations, which we firmly believed was the optimal solution. Only a few months later, when the algorithm presented here was implemented, the computer proved us wrong when it produced just 25 equations.

Numadd Our third example is a function that operates on a representation of arithmetic expressions and is used as part of a decision procedure for Presburger arithmetic. Figure 4.5 shows the pattern matching used here. Unfortunately, in this example the minimization brings no improvement over TFL's disambiguation: The set of 256 (!) resulting equations is already minimal.

4.3.5 Discussion

Our implementation performs reasonably well on problems from Isabelle theories. There are however two issues: First, there exist examples (such as *Numadd* above) where the blowup is unavoidable, and the mininimization does not save us from getting overly large equational specifications. Second, for larger examples, the computational complexity of the problem might make exact minimization practically intractable. Then, one could try to apply heuristic methods, and we expect the methods developed for boolean minimization to be applicable in this context.

Alternative encodings

Given the fact that minimization cannot always ensure a short disambiguation, it seems that the only definite solution to the pattern explosion problem is to find alternative logical representations for overlapping patterns, instead of instantiating them to remove the overlap. For example, the second equation of the *sep* function in Fig. 4.1(a) could also be expressed as a conditional equation with quantifiers:

$$(\bigwedge x\ y\ ys.\ xs \neq (x : y : ys)) \implies sep\ a\ xs = xs$$

In general, any equation would have preconditions that ensure that no earlier pattern matches the given input. With this construction, the increase in specification size would be quadratic in the worst case instead of exponential.

With proper tool support it is also imaginable to use the definitions using *MATCH* directly for reasoning — as a "native" representation of clausal function definitions in higher-order logic.

However, the extra complexity introduced by such a construction makes proofs more technical and destroys the purely equational view, which is very common in the informal proofs from the literature on functional programming (see e.g. Hughes [57] or Thompson [105]). In particular, all automated reasoning tools would need to be adapted to deal with such a representation.

For this reason, we focused on the optimization opportunities (and their limits) on purely equational specifications at this point, for which appropriate reasoning tools are already in place.

Open questions

Although our main motivation is to reason about functional programs in a theorem prover, our results might be relevant for other areas, since they naturally generalize results on boolean minimization.

Disambiguating an equational specification into independent equations could be of use in a form of parallel pattern matching, where several patterns are matched simultaneously (say, in different threads). This form of parallel matching differs from the usual meaning of the term, where different components of the *same* pattern are matched concurrently, in order to let the match fail when any of the components fails to match [116]. In contrast, unambigous pattern sets in our sense would allow to compute the matches of *different* equations in parallel. In a lazy language, prime implicants, being the most general patterns describing a certain set of values, would probably play an important role, since they are general enough not to trigger an unnecessary evaluation.

These are however just vague ideas and it is not clear if they could be exploited to make some improvement in the area of implementation of functional languages.

Another interesting question is whether one can also do the reverse encoding of §4.3.2, i.e. encode a given pattern minimization problem into a boolean formula and read off the solution from the minimized formula. Then we could readily use existing high-performance boolean minimizers (e.g. Espresso [24]) to solve our pattern problems. From Σ_2^P-completeness, we know that there must be such an encoding, but we could not find a natural one. The main problem here is that the pattern problems contain more structure, which may be destroyed during boolean minimization.

4.4 Related Work

Encoding pattern matching in terms of other language elements is also possible in functional languages, and a beautiful description of such an encoding is given by Rhiger [93]. Our encoding is different in that it is not executable, but uses the description operator, which allows a more general class of patterns.

Although the pattern minimization problem we consider is simple to state and natural, it has, up to our knowledge, never been studied systematically. The only complexity result related to ML-like pattern matching that we are aware of is given in an unpublished extendend abstract by Baudinet and MacQueen [10]. It states that the problem of compiling a sequence of patterns in to a decision tree (i.e. a case expression) of minimal size is NP-complete. Our transformation is different, since it produces a set of equations again, and not a decision tree. It is remarkable that this puts the problem into a different complexity class.

There has been a fair amount of research on pattern match compilation [5, 110, 37], but it is hard to compare this work with ours, since it has the goal of producing code that can be implemented efficiently, either in the form of a case tree or of some kind of backtracking automaton. Our optimization problem is different, due to the focus on the equational view.

Chapter 5

Induction Schemes

Contents

5.1 User-Specified Induction Rules

In this last and rather short chapter, we put the previously developed methods into a slightly different context, and show that parts of the machinery developed for function definitions and termination proofs can also be used for proving user-specified induction principles automatically.

The function package already derives induction rules for function definitions, which are built to mirror the recursive structure of the function that was extracted by the mechanism described in §2.5. This often works well, but in some cases the generated rules are not optimal for certain tasks and a custom rule would be more convenient or more powerful.

In the following we present a proof procedure that derives a wide class of induction rules automatically from simpler parts. It generalizes the process that is used internally in the function package, but since it is also useful independently, we make it available to the user as a method called *induct-scheme*.

It is not uncommon that users of Isabelle derive their own induction rules if the rules provided by packages are suboptimal for some reason. Working in a higher-order logic simplifies this task, since induction rules are just normal theorems instead of axiom schemes that must be justified extra-logically, as in first-order logic. However, proving induction rules by hand is difficult because of their complex structure: Three levels of nested implications are not unusual, and a proof of an induction rule typically relies on another induction rule, possibly instantiating it with a structured formula. This can be a daunting task for beginners, and even experts must stare at the nested arrows for a while before they get the proof right. The task is further complicated by the lack of automation, since the standard tools are not very helpful with proofs of this type.

The idea of *induct-scheme* is to reduce a goal that states an induction rule to simpler goals that express the different aspects of the induction: case distinction, wellfoundedness, and invariant preservation. This has two advantages: First, the resulting proof obligations are structurally simpler and easier to prove than the original one. Second, and more importantly, they have a form that is much better suited for automation, and which can often be solved fully automatically.

Unlike the function package, which extracts recursive calls from a function in a heuristic manner that requires careful configuration, *induct-scheme* has no heuristic component. It merely mechanizes a transformation that is well-defined and entirely predictable.

5.2 The General Format

The general format of the induction rules that we can derive is given by the following schema[1]:

$$\frac{\begin{array}{l}\Delta_1 \Longrightarrow (\Gamma_{1,1} \Longrightarrow P\ r_{1,1}) \Longrightarrow \cdots \Longrightarrow (\Gamma_{1,m_1} \Longrightarrow P\ r_{1,m_1}) \Longrightarrow P\ p_1 \\ \vdots \qquad\qquad \vdots \\ \Delta_n \Longrightarrow (\Gamma_{n,1} \Longrightarrow P\ r_{n,1}) \Longrightarrow \cdots \Longrightarrow (\Gamma_{n,m_n} \Longrightarrow P\ r_{n,m_n}) \Longrightarrow P\ p_n\end{array}}{I\ x \Longrightarrow P\ x}$$

Note that only x and the induction predicate P in the schema above are logical variables. The other symbols are schema variables that represent the terms or contexts that appear in a concrete rule. However, the contexts Δ_i and $\Gamma_{i,j}$ often contain other variables. The induction has n cases that correspond to its premises. Each case consists of a context Δ_i, a conclusion $P\ p_i$, and a set of induction hypotheses, each possibly with its own context. There may also be global conditions $I\ x$, which act like an invariant and constrain the induction to a subset of the whole type. Syntactically, $I\ x$ is a (possibly empty) list of conditions that refer to x. It is not a context, since it cannot bind new variables.

This already looks quite complicated, but it is not the whole story yet, since it describes only rules with a single induction predicate that takes a single argument. Induction rules with multiple induction predicates of arity greater than one will be discussed later, but they add only technical complexity and nothing fundamentally new.

To get a feel for the rule schema, let us look at some small instances:

Example 5.1. Here is the well-known structural induction for natural numbers:

$$\frac{P\ 0 \qquad \bigwedge n.\ P\ n \Longrightarrow P\ (\mathit{Suc}\ n)}{P\ n}$$

The first case has an empty context Δ_1, no recursive call and the pattern 0. In the second, Δ_2 binds a variable n, $p_2 = \mathit{Suc}\ n$, and there is one recursive call $r_{2,1} = n$ whose context $\Gamma_{2,1}$ is empty. There are no global conditions.

[1] After careful consideration of alternatives, we use the following slightly subtle terminology: Following the crowd in automated reasoning, we employ the term *induction scheme* to refer to an induction rule, i.e., a theorem with a predicate variable. In contrast, the term *schema* refers to a description of the syntactic format of such rules.

Example 5.2. The following induction rule shows a property for all even numbers:

$$\frac{P\ 0 \qquad \bigwedge n.\ \mathit{even}\ n \Longrightarrow P\ n \Longrightarrow P\ (\mathit{Suc}\ (\mathit{Suc}\ n))}{\mathit{even}\ x \Longrightarrow P\ x}$$

Here, the global condition $I\ x$ is *even* x. We have a zero case and a case for *Suc* (*Suc* n), and the context of the second case, Δ_2, now carries an extra assumption *even* n.

5.2.1 Proof obligations

To derive an instance of the schema we have presented, the *induct-scheme* method demands a proof of three proof obligations, two of which are familiar from the previous chapters:

Case completeness Case completeness states that the cases in the induction rule cover all values:

$$\frac{I\ x \qquad \Delta_1 \Longrightarrow x = p_1 \Longrightarrow R \qquad \ldots \qquad \Delta_n \Longrightarrow x = p_n \Longrightarrow R}{R}$$

This is exactly the completeness condition that we have used in §4.2 to define functions with pattern matching, except for the additional premise $I\ x$.

The compatibility condition that was needed for the consistency of function definitions is not relevant for induction rules, since patterns may overlap freely.

Wellfoundedness Wellfoundedness ensures that there is no circularity in the induction. For every induction hypothesis, we generate the relation

$$C_{i,j} = \{(r_{i,j}, p_i) \mid \Delta_i \wedge \Gamma_{i,j}\}\ .$$

The proof obligation is that the union of these relations is wellfounded:

$$\mathit{wf}\ (C_{1,1} \cup \cdots \cup C_{1,m_1} \cup \cdots \cup C_{n,1} \cup \cdots \cup C_{n,m_n})$$

As another déjà vu, this proof obligation is a termination goal in closed form (c.f. §3.1.1), and we have already worked for a whole chapter on solving this class of goals automatically.

Invariant preservation If the induction rule is conditional, we must make sure that the condition is also true for the arguments of the induction hypothesis:

$$\Delta_i \Longrightarrow \Gamma_{i,j} \Longrightarrow I\ r_{i,j}$$

Since I may consist of multiple conditions, it is nicer to use the elimination rule format, which keeps it on the left-hand side of the arrow:

$$\Delta_i \Longrightarrow \Gamma_{i,j} \Longrightarrow (I\ r_{i,j} \Longrightarrow R) \Longrightarrow R$$

Example 5.3. For the structural induction rule for *nat*, the following proof obligations are generated:

Case completeness:

$$\frac{x = 0 \Longrightarrow P \qquad \bigwedge n.\ x = Suc\ n \Longrightarrow P}{P}$$

Wellfoundedness:

$$wf\ \{(n,\ Suc\ n)\ |n.\ True\}$$

Invariant preservation:

$$\bigwedge n.\ R \Longrightarrow R$$

Modulo variable renaming, the case completeness condition is identical with the case distinction rule for the datatype *nat*. Wellfoundedness is a very simple termination proof obligation, and the invariant preservation condition is vacuous and not even presented to the user.

Example 5.4. For our second example above, the induction rule about *even*, we must prove the following conditions:

Case completeness:

$$\frac{even\ x \qquad x = 0 \Longrightarrow P \qquad \bigwedge n.\ x = Suc\ (Suc\ n) \Longrightarrow P}{P}$$

Wellfoundedness:

$$wf\ \{(n,\ Suc\ (Suc\ n))\ |n.\ even\ n\}$$

Invariant preservation:

$$\bigwedge n.\ even\ n \Longrightarrow (even\ n \Longrightarrow R) \Longrightarrow R$$

Case completeness states that any even number is either zero or the successor of the successor of an even number. If the *even* predicate was defined inductively, this is just its elimination rule. Wellfoundedness is unexciting and the invariant preservation is again trivial, although we have an invariant this time. But the context Δ_2 explicitly states that n (the argument of the induction hypothesis) is even, so again we have nothing to prove.

These small examples are a little boring, since they just derive very simple rules that are present in the system anyway. However, the power of the *induct-scheme* method lies in its flexibility concerning the actual syntactic form of the rule.

Example 5.5. Let us derive an induction rule over type α *list list*. In the induction step, we would like to have a hypothesis for shorter lists and one for lists whose first sublist is shorter and the other lists are unchanged:

$$\frac{\begin{array}{l} P\ [\,] \\ \bigwedge xs\ xss. \\ \quad (\bigwedge yss.\ list\text{-}size\ yss < list\text{-}size\ (xs{:}xss) \Longrightarrow P\ yss) \Longrightarrow \\ \quad (\bigwedge zs.\ list\text{-}size\ zs < list\text{-}size\ xs \Longrightarrow P\ (zs{:}xss)) \Longrightarrow \\ \quad P\ (xs{:}xss) \end{array}}{P\ a}$$

Pattern completeness is again just a property of the list datatype, preservation is not required, and the wellfoundedness proof obligation is the following:

$$\begin{aligned}&\mathit{wf}\ (\{(\mathit{yss},\ \mathit{xs}{:}\mathit{xss})\ |\mathit{xs}\ \mathit{xss}\ \mathit{yss}.\ \mathit{list\text{-}size}\ \mathit{yss} < \mathit{list\text{-}size}\ (\mathit{xs}{:}\mathit{xss})\} \cup \\ &\quad \{(\mathit{zs}{:}\mathit{xss},\ \mathit{xs}{:}\mathit{xss})\ |\mathit{xs}\ \mathit{xss}\ \mathit{zs}.\ \mathit{list\text{-}size}\ \mathit{zs} < \mathit{list\text{-}size}\ \mathit{xs}\})\end{aligned}$$

It is proved automatically by *lexicographic-order*, using a lexicographic combination of the measures *list-size* and *gen-list-size list-size*. Thus, the complete proof of this custom induction rule consists of a single line, which is idiomatic for proofs of induction rules:

by *induct-scheme* (*pat-completeness*, *lexicographic-order*)

5.3 Internal Derivation

We now sketch how the method works internally. After analyzing the goal to find the instantiation of the schema variables, it builds the relation

$$R = C_{1,1} \cup \cdots \cup C_{1,m_1} \cup \cdots \cup C_{n,1} \cup \cdots \cup C_{n,m_n}$$

which is the basis of the following induction proof, that derives the induction rule from the case completeness, wellfoundedness and invariant preservation conditions:

Proof. We assume the premises that correspond to the cases; let us call them the *step rules*. The implication $I\ x \Longrightarrow P\ x$ is proved by wellfounded induction over x using the relation R, which is wellfounded by assumption.

For a fixed x, we assume $I\ x$ and the induction hypothesis

$$\bigwedge z.\ (z,\ x) \in R \Longrightarrow I\ z \Longrightarrow P\ z.$$

Our goal is to show $P\ x$. We apply the case completeness condition as an elimination rule and obtain n cases. In the ith case, the assumptions are Δ_i and $x = p_i$, and we must prove $P\ p_i$.

We now consider the recursive calls in turn. From the construction of R, it follows that $\Gamma_{i,j} \Longrightarrow (r_{i,j},\ p_i) \in C_{i,j} \subseteq R$, and the induction hypothesis implies $\Gamma_{i,j} \Longrightarrow I\ r_{i,j} \Longrightarrow P\ r_{i,j}$. The invariant preservation condition implies that we can remove the condition $I\ r_{i,j}$. We repeat this for every call and can finally apply the step rule to obtain $P\ p_i$, which completes the proof of the ith case. □

5.4 Example Applications

This section presents some interesting applications of the *induct-scheme* method, which shows how it can simplify the life of Isabelle users.

5.4.1 Huffman's algorithm: Consistent trees

Our first example comes from Blanchette's formalization of the textbook proof of Huffman's algorithm [17]. His primary datatype is a binary tree:

datatype α *tree* = *Leaf nat* α
  | *InnerNode nat* (α *tree*) (α *tree*)

$$\frac{\begin{array}{l}
\bigwedge w_b\ b\ a.\ P\ (\mathit{Leaf}\ w_b\ b)\ a \\[1ex]
\bigwedge w\ t_1\ t_2\ a. \\
\quad \mathit{consistent}\ t_1 \Longrightarrow \\
\quad \mathit{consistent}\ t_2 \Longrightarrow \\
\quad \mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \emptyset \Longrightarrow \\
\quad a \in \mathit{alphabet}\ t_1 \Longrightarrow \\
\quad a \notin \mathit{alphabet}\ t_2 \Longrightarrow \\
\quad P\ t_1\ a \Longrightarrow P\ t_2\ a \Longrightarrow P\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a \\[1ex]
\bigwedge w\ t_1\ t_2\ a. \\
\quad \mathit{consistent}\ t_1 \Longrightarrow \\
\quad \mathit{consistent}\ t_2 \Longrightarrow \\
\quad \mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \emptyset \Longrightarrow \\
\quad a \notin \mathit{alphabet}\ t_1 \Longrightarrow \\
\quad a \in \mathit{alphabet}\ t_2 \Longrightarrow \\
\quad P\ t_1\ a \Longrightarrow P\ t_2\ a \Longrightarrow P\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a \\[1ex]
\bigwedge w\ t_1\ t_2\ a. \\
\quad \mathit{consistent}\ t_1 \Longrightarrow \\
\quad \mathit{consistent}\ t_2 \Longrightarrow \\
\quad \mathit{alphabet}\ t_1 \cap \mathit{alphabet}\ t_2 = \emptyset \Longrightarrow \\
\quad a \notin \mathit{alphabet}\ t_1 \Longrightarrow \\
\quad a \notin \mathit{alphabet}\ t_2 \Longrightarrow \\
\quad P\ t_1\ a \Longrightarrow P\ t_2\ a \Longrightarrow P\ (\mathit{InnerNode}\ w\ t_1\ t_2)\ a
\end{array}}{\mathit{consistent}\ t \Longrightarrow P\ t\ a}$$

Figure 5.1: Induction rule for consistent trees

A predicate *consistent* :: α *tree* $\Rightarrow$ *bool* models trees that satisfy a certain invariant. The whole algorithm works only with consistent trees and many relevant properties fail if this invariant is violated.

To simplify proofs by structural induction over consistent trees, Blanchette uses the custom induction rule given in Fig. 5.1. This rule has two advantages over the standard induction rule: First, the assumption *alphabet* $t_1 \cap$ *alphabet* $t_2 = \emptyset$, which follows from consistency, is immediately available in each case. Second, an extra case distinction is built into the rule to distinguish whether the symbol a occurs in the left, the right, or none of the subtrees. The case that it occurs in both is missing — it is impossible for consistent trees.

This rule (and another similar one) drastically simplifies the rest of the development, since the reasoning would otherwise have to be done manually in each of the numerous induction proofs. Using the custom rule, many of these proofs become one-liners.

The induction rule itself is easy to prove, too: After applying *induct-scheme*, only a single case distinction must be done manually for the case completeness. The other conditions are solved automatically by *lexicographic-order* and *fastsimp*. In contrast, the manual proof of the same rule takes several pages.

$$\dfrac{\begin{array}{l}\bigwedge S.\ \mathit{invariant}\ S \Longrightarrow P\ S\ [\,] \\ \bigwedge S\ x\ \mathit{xs}. \\ \quad \mathit{invariant}\ S \Longrightarrow \\ \quad \mathit{is\text{-}node}\ x \Longrightarrow \\ \quad \forall\, y{\in}\mathit{set}\ \mathit{xs}.\ \mathit{is\text{-}node}\ y \Longrightarrow \\ \quad (\mathit{memb}\ x\ S \Longrightarrow P\ S\ \mathit{xs}) \Longrightarrow \\ \quad (\neg\ \mathit{memb}\ x\ S \Longrightarrow P\ (\mathit{ins}\ x\ S)\ (\mathit{succs}\ x\ @\ \mathit{xs})) \Longrightarrow P\ S\ (x{:}\mathit{xs})\end{array}}{\mathit{invariant}\ S \Longrightarrow \forall\, x{\in}\mathit{set}\ \mathit{xs}.\ \mathit{is\text{-}node}\ x \Longrightarrow P\ S\ \mathit{xs}}$$

Figure 5.2: Induction rule for depth-first search

5.4.2 Depth-first search

We consider again the abstract version of depth-first search that we already saw in §2.7.4. Although an induction principle for *dfs* can be obtained from the partial induction rule that arises from the definition, it is actually simpler to prove it straight away using *induct-scheme*. This gives the best control over the form of the rule, which is given in Fig. 5.2.

In this case the proof is not automated, and the wellfoundedness condition is the hardest part. It corresponds to a termination proof of *dfs* for any well-formed input.

5.4.3 Strong nominal induction

The nominal datatype package [108] provides an advanced infrastructure for reasoning about data structures that involve binders, e.g., terms in the lambda calculus. When modelling lambda terms as a nominal datatype, terms that differ only in the naming of bound variables are identified.

The induction rule for nominal datatypes is stronger than the normal datatype rule: In the lambda case, we may assume that the name of the bound variable is fresh with respect to a freshness context z which we may choose when we apply the rule. This freshness context formalizes what is known as the variable convention and avoids name clashes that would require explicit renaming:

$$\dfrac{\begin{array}{l}\bigwedge v\ z.\ P\ z\ (\mathit{Var}\ v) \\ \bigwedge s\ t\ z.\ (\bigwedge z.\ P\ z\ s) \Longrightarrow (\bigwedge z.\ P\ z\ t) \Longrightarrow P\ z\ (\mathit{App}\ s\ t) \\ \bigwedge v\ t\ z.\ v \sharp z \Longrightarrow (\bigwedge z.\ P\ z\ t) \Longrightarrow P\ z\ (\mathit{Lam}\ [v].t)\end{array}}{P\ z\ t}$$

The nominal package derives this rule automatically for each datatype in a nontrivial automated proof. However, the rule also conforms to our schema, so we can use *induct-scheme* to prove it. This leads to the following proof obligations:

$$\dfrac{\begin{array}{l}\bigwedge v\ \mathit{za}.\ z = \mathit{za} \Longrightarrow t = \mathit{Var}\ v \Longrightarrow P \\ \bigwedge s\ \mathit{ta}\ \mathit{za}.\ z = \mathit{za} \Longrightarrow t = \mathit{App}\ s\ \mathit{ta} \Longrightarrow P \\ \bigwedge v\ \mathit{ta}\ \mathit{za}.\ v \sharp \mathit{za} \Longrightarrow z = \mathit{za} \Longrightarrow t = \mathit{Lam}\ [v].\mathit{ta} \Longrightarrow P\end{array}}{P}$$

$$\begin{aligned}&\mathit{wf}\ (\{((zb,\ s),\ (za,\ \mathit{App}\ s\ ta))\ |s\ ta\ za\ zb.\ \mathit{True}\} \cup\\ &\quad \{((zb,\ ta),\ (za,\ \mathit{App}\ s\ ta))\ |s\ ta\ za\ zb.\ \mathit{True}\} \cup\\ &\quad \{((zb,\ ta),\ (za,\ \mathit{Lam}\ [v].ta))\ |v\ ta\ za\ zb.\ v \sharp za\})\end{aligned}$$

The case completeness condition is a form of strong elimination rule for the datatype, which also has the freshness context, but is conceptually simpler, since there is no induction involved. The inductive argument is captured by the wellfoundedness condition, which is simple and can be proved automatically, provided that a size function on lambda-terms is present.

As the nominal package already provides this specific induction rule, there is no need for proving it again, but the example shows the generality of our schema. Moreover, as soon as a small variation of the basic rule is needed, e.g., a simultaneous induction over two terms, then *induct-scheme* may turn out to be useful.

5.5 Multiple Induction Predicates

We now give the extended schema for rules with multiple induction predicates $P_1, \dots, P_k$ taking multiple arguments each. To keep the presentation readable, we write the arguments as a vector $\bar{x}_i$:

$$\frac{\begin{array}{l}\Delta_1 \Longrightarrow (\Gamma_{1,1} \Longrightarrow P_{j_{1,1}}\ \bar{r}_{1,1}) \Longrightarrow \cdots \Longrightarrow (\Gamma_{1,m_1} \Longrightarrow P_{j_{1,m_1}}\ \bar{r}_{1,m_1}) \Longrightarrow P_{i_1}\ \bar{p}_1\\ \vdots \qquad\qquad \vdots\\ \Delta_n \Longrightarrow (\Gamma_{n,1} \Longrightarrow P_{j_{n,1}}\ \bar{r}_{n,1}) \Longrightarrow \cdots \Longrightarrow (\Gamma_{n,m_n} \Longrightarrow P_{j_{n,m_n}}\ \bar{r}_{n,m_n}) \Longrightarrow P_{i_n}\ \bar{p}_n\end{array}}{(I_1\ \bar{x}_1 \Longrightarrow P_1\ \bar{x}_1) \qquad \dots \qquad (I_k\ \bar{x}_k \Longrightarrow P_k\ \bar{x}_k)}$$

This rule has one conclusion for each induction predicate.[2] The different predicates may range over different types and have their own conditions $I_1, \dots, I_k$ attached to them. Each of the cases has one of the induction predicates in the conclusion, but its inductive hypotheses can refer to other induction predicates.

Example 5.6. The following induction rule is a simple instance of the extended schema above:

$$\frac{\begin{array}{l}P\ 0\\ Q\ 0\\ \bigwedge n.\ P\ n \Longrightarrow Q\ (\mathit{Suc}\ n)\\ \bigwedge n.\ Q\ n \Longrightarrow P\ (\mathit{Suc}\ n)\end{array}}{P\ x \qquad Q\ x}$$

Introducing multiple induction predicates and multiple variables is a conceptually straightforward extension, as they are easily encoded using products (for multiple variables) and sums (for multiple predicates), and the induction proof presented in §5.3 works in essentially the same way, with some more bookkeeping. The reader will notice the similarity to the treatment of curried and mutually recursive function definitions (§2.6.4); in fact, the example above is the induction rule for the mutually recursive definition of *even* and *odd*.

[2]We interpret multiple conclusions as a conjunction. In Isabelle, a rule with multiple conclusions is expressed as a set of rules.

The proof obligations produced by *induct-scheme* for this class of rules are similar to the previous ones: Instead of one, we now get k proof obligations for case completeness, each of them asserting that the cases for the respective induction predicate are complete. In the wellfoundedness condition, the internal representation is exposed, and we must prove wellfoundedness of a relation over sums of products — a proof obligation as it arises from mutually recursive functions. The invariant preservation conditions are as expected: For each recursive call we must prove the invariant of the respective induction predicate.

Chapter 6

Conclusion

The results in this thesis are centered around the goal of making function definitions in Isabelle/HOL easier, more powerful, and more **fun**, while retaining the safety of the definitional approach.

6.1 New Toys

The implementation of the techniques we have described is now available to users of Isabelle as a collection of commands and proof methods, which give a nice summary of our results from a practical viewpoint:

- A command *function*, which introduces recursive functions and supports all complications: partiality, nested recursion, mutual recursion, general pattern matching, and tail recursion. Compared to existing approaches, we have increased both the expressive power (by supporting partial functions and more general pattern matching) and the convenience in formal reasoning (by supporting nested recursive definitions naturally).

- A command *fun*, which invokes *function*, followed by automated methods to solve the pattern matching and termination proof obligations. This command provides a convenient shorthand notation that provides full automation for the most common cases.

- A method *lexicographic-order*, which solves termination proof obligations using lexicographic combinations of measure functions. The method is sufficient for a large majority of the termination problems occurring in theorem proving practice.

- A method *sizechange*, which implements a stronger termination prover using dependency graph decomposition and the size-change principle. It can handle functions with more complicated control and data flow.

- A method *induct-scheme*, which proves induction rules from simpler proof obligations that are better suited for automation, which considerably simplifies proofs of custom induction rules.

Our implementation is the principal tool for working with general recursive functions in Isabelle/HOL since version 2007. It is documented in a tutorial [63].

It should be mentioned that the improved automation is not only helpful for users, but also facilitates the implementation of other tools. For instance, translating function definitions from other languages into Isabelle theories is now much easier, since the function package hides the details of the construction from the translation tools. In particular, no termination relation has to be specified for the definitions[1]. Two examples of such translation tools are Haskabelle[52], which converts Haskell programs into Isabelle theories, and Ott [96], which translates language specifications from a custom notation to various formalisms, including Isabelle/HOL theories. Both tools use the function package for making definitions and rely on the automated termination prover.

Most of the methods that we have described are not specific to Isabelle, and our techniques for proving termination (especially the simple approach of *lexicographic-order*) are likely to be adopted by other systems, since they can be implemented with relatively little effort.

Open Problems However, some issues remain and recursive function definitions will continue to be difficult in some cases:

- The pattern matching explosion we discussed in §4.3 is an obstacle for the verification of programs that make heavy use of complicated pattern matching. As we have shown, minimizing the number of equations is expensive and does not help in some cases.
- The discrepancy between the domain predicate and actual termination behaviour is at least unusual. In particular, when generating code, reasoning about the termination behaviour in the target language is impossible.
- The extraction mechanism for recursive calls (§2.5) cannot cope with all definitions, and definitions that use some advanced of higher-order programming techniques (like certain forms of continuation-passing style) cannot be supported.

All these issues are ultimately rooted in the subtle semantic differences between higher-order logic and programming languages. After all, the slogan *HOL = Functional Programming + Logic* must be taken with a grain of salt.

Some of these issues could be overcome at least partly: The pattern matching explosion can be avoided by moving away from purely equational specifications and using conditional equations as we sketched in §4.3.5. By using a fixed set of congruence rules and making undefinedness propagate, the domain predicate could be modified to agree with the termination behaviour for a fixed set of constructs (say, *if*, *let*, and *case*) in some fixed target language. However, this would sacrifice the generality of the approach. To support all function definitions with no exceptions, one must probably move to an entirely different representation, e.g., HOLCF [77].

[1]Not surprisingly, such a translation is inherently incomplete unless it is known that all translated functions can be handled by the termination prover. However, the user can always add a manual termination proof later if necessary.

6.2 Future Work

We now briefly discuss some ideas for future work:

Recursion in a monad Monads are commonly used in Haskell to model certain computational aspects of programs, such as updatable state, exceptions, nondeterminism, and other nontrivial control flow behaviour [91]. Although general monads cannot be expressed in HOL due to the lack of constructor polymorphism, specific monads are becoming a popular modeling device [100, 25].

Defining functions over a monadic type can pose new difficulties for recursive function definitions. For example, consider the function *traverse*, which traverses a list stored on a monadic heap:

```
datatype α node = Empty | Node α (α node ref)

traverse :: α node ⇒ α list Heap
traverse Empty = return []
traverse (Node x r) = do tl ← !r;
                         xs ← traverse tl;
                         return (x:xs)
```

Since the recursion does not happen on the arguments of the function but rather on an encapsulated state, the function package cannot support it. To define the function we must break the monad abstraction and expose the underlying state. Furthermore, the function is partial, since the list on the heap may be infinite or cyclic.

However, there is also a simple and elegant solution, which is based on the fact that in a monad with a left zero (that is, an element 0 that satisfies $(0 \gg= f) = 0$), there is a total model for *any* recursive function. This is because the partial function can be completed with 0, which propagates correctly through the monadic computations.

Using the default value facility (§2.6.1), we have already done this construction manually for individual functions. However, automating it in a package would make working with monadic functions much easier than it is now.

Infinite data structures We already mentioned that our approach with well-founded recursion cannot be used to define functions on infinite data structures like lazy lists and streams, which would require some form of corecursion. Matthews [70] describes a construction using converging equivalence relations, but it has never been automated. Tools that support such definitions would make coinductive data structures much more attractive for formalizations.

Termination Our termination prover deals well with many termination problems, and the road becomes steep if we want to improve it further. It seems that the highest practical benefit could be drawn from using polynomial interpretation on data types. It is possible that methods developed for term rewrite systems can be adapted to the Isabelle setting. The main obstacle here is that finding a polynomial interpretation requires an analysis of the goals, which is hard due to their general form.

Other representations for patterns Representing pattern matching via conditional equations is probably the cleanest solution to the pattern problem, but it requires that the proof tools of Isabelle/HOL can work well on this representation. It should be investigated what changes are required for the simplifier to make such a representation practical.

Bibliography

[1] Archive of Formal Proofs. http://afp.sourceforge.net/.

[2] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science Volume 3*, pages 1–168. Oxford University Press, 1994.

[3] O. Ait Mohamed, C. Muñoz, and S. Tahar, editors. *Theorem Proving in Higher Order Logics (TPHOLs 2008), 21st International Conference, Montreal, Canada, August 18-21, 2008, Proceedings*, volume 5170 of *Lecture Notes in Computer Science*. Springer Verlag, 2008.

[4] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[5] L. Augustsson. Compiling pattern matching. In *Functional Programming Languages and Computer Architecture (FPCA'85)*, volume 201 of *Lecture Notes in Computer Science*, pages 368–381. Springer Verlag, 1985.

[6] J. Avery. Size-change termination and bound analysis. In Hagiya and Wadler [50], pages 192–207.

[7] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[8] C. Ballarin. Locales and locale expressions in isabelle/isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs (TYPES 2003)*, volume 3085 of *Lecture Notes in Computer Science*, pages 34–50. Springer Verlag, 2004.

[9] G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In Hagiya and Wadler [50], pages 114 – 129.

[10] M. Baudinet and D. MacQueen. Tree pattern matching for ML. Unpublished. http://www.smlnj.org/compiler-notes/85-note-baudinet.ps, 1985.

[11] A. Ben-Amram and M. Codish. A SAT-based approach to size change termination with global ranking functions. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963 of *Lecture Notes in Computer Science*, pages 218–232. Springer Verlag, March 2008.

[12] A. M. Ben-Amram and C. S. Lee. Ranking functions for size-change termination II. July 2007. International Workshop on Termination (WST'07).

[13] S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, volume 2277 of *Lecture Notes in Computer Science*, pages 24–40. Springer Verlag, 2000.

[14] S. Berghofer and M. Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In Bertot et al. [16], pages 19–36.

[15] Y. Bertot and P. Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.* Texts in theoretical computer science. Springer Verlag, 2004.

[16] Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors. *Theorem Proving in Higher Order Logics (TPHOLs '99)*, volume 1690 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[17] J. C. Blanchette. Mechanizing the textbook proof of Huffman's algorithm. *Journal of Automated Reasoning*, 2009. To appear.

[18] F. Blanqui, S. Coupet-Grimal, W. Delobel, S. Hinderer, and A. Koprowski. CoLoR, a Coq library on rewriting and termination. In A. Geser and H. Söndergaard, editors, *International Workshop on Termination (WST'06)*, 2006.

[19] A. Bove. General recursion in type theory. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *Lecture Notes in Computer Science*, pages 39–58. Springer Verlag, 2002.

[20] A. Bove and V. Capretta. Nested general recursion and partiality in type theory. In R. J. Boulton and P. B. Jackson, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2001)*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer Verlag, 2001.

[21] A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

[22] A. Bove and V. Capretta. Computation by prophecy. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications (TLCA 2007)*, volume 4583 of *Lecture Notes in Computer Science*, pages 70–83. Springer Verlag, 2007.

[23] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[24] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, Boston, MA, 1984.

[25] L. Bulwahn, A. Krauss, F. Haftmann, L. Erkök, and J. Matthews. Imperative functional programming in Isabelle/HOL. In Ait Mohamed et al. [3], pages 134–149.

[26] L. Bulwahn, A. Krauss, and T. Nipkow. Finding lexicographic orders for termination proofs in Isabelle/HOL. In Schneider and Brandt [95], pages 38–53.

[27] A. Chaieb. Verifying mixed real-integer quantifier elimination. In Furbach and Shankar [39], pages 528–540.

[28] A. Chaieb. *Automated methods for formal proofs in simple arithmetics and algebra*. PhD thesis, Institut für Informatik, Technische Universität München, Germany, January 2008.

[29] A. Church. A formulation of the simple theory of types. *J. Symbolic Logic*, pages 56–68, 1940.

[30] M. Codish, V. Lagoon, and P. J. Stuckey. Solving partial order constraints for LPO termination. In Pfenning [92], pages 4–18.

[31] E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In B. Konev and F. Wolter, editors, *Frontiers of Combining Systems (FroCos 07)*, volume 4720 of *Lecture Notes in Artificial Intelligence*, pages 148–162, Liverpool,UK, Sept. 2007. Springer Verlag.

[32] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In M. I. Schwartzbach and T. Ball, editors, *Programming Language Design and Implementation (PLDI'06)*, pages 415–426. ACM, 2006.

[33] P. Courtieu, J. Forest, and X. Urbain. Certifying a termination criterion based on graphs, without graphs. In Ait Mohamed et al. [3], pages 183–198.

[34] C. Dubois and V. Donzeau-Gouge. A step towards the mechanization of partial functions: domains as inductive predicates. In *CADE-15 Workshop on mechanization of partial functions*, 1998.

[35] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *J. Symbolic Logic*, 65(2):525–549, 2000.

[36] N. Een and N. Sörensson. An extensible SAT-solver. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Theory and Applications of Satisfiability Testing (SAT 2003)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502 – 518. Springer Verlag, 2003.

[37] F. L. Fessant and L. Maranget. Optimizing pattern matching. In *ICFP 2001*, pages 26–37, 2001.

[38] S. Finn, M. Fourman, and J. Longley. Partial functions in a total setting. *Journal of Automated Reasoning*, 18(1):85–104, 1997.

[39] U. Furbach and N. Shankar, editors. *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Artificial Intelligence*. Springer Verlag, 2006.

[40] J. Giesl. Termination analysis for functional programs using term orderings. In A. Mycroft, editor, *Static Analysis (SAS'95)*, volume 983 of *Lecture Notes in Computer Science*, pages 154–171. Springer Verlag, 1995.

[41] J. Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19(1):1–29, Aug. 1997.

[42] J. Giesl. Induction proofs with partial functions. *Journal of Automated Reasoning*, 26(1):1–49, 2001.

[43] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In Furbach and Shankar [39], pages 281–286.

[44] M. Gordon and T. Melham, editors. *Introduction to HOL: A theorem proving environment for Higher Order Logic*. Cambridge University Press, 1993.

[45] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[46] B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In J. Hurd and T. F. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer Verlag, 2005.

[47] J. Grundy and M. C. Newey, editors. *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98, Canberra, Australia, September 27 - October 1, 1998, Proceedings*, volume 1479 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[48] F. Haftmann and T. Nipkow. A code generator framework for Isabelle/HOL. Technical Report 364/07, Department of Computer Science, University of Kaiserslautern, 08 2007.

[49] F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs (TYPES 2006)*, volume 4502 of *Lecture Notes in Computer Science*. Springer Verlag, 2007.

[50] M. Hagiya and P. Wadler, editors. *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, volume 3945 of *Lecture Notes in Computer Science*. Springer Verlag, 2006.

[51] J. Harrison. The HOL Light theorem prover. http://www.cl.cam.ac.uk/users/~jrh13/hol-light.

[52] Haskabelle tool. http://isabelle.in.tum.de/haskabelle.

[53] N. Hirokawa and A. Middeldorp. Tyrolean termination tool. In J. Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 175–184. Springer Verlag, 2005.

[54] P. V. Homeier and D. F. Martin. Mechanical verification of total correctness through diversion verification conditions. In Grundy and Newey [47], pages 189–206.

[55] P. Hudak, J. Hughes, S. L. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. pages 1–55. ACM, 2007.

[56] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.

[57] J. Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[58] J. Hurd. Proof pearl: The termination analysis of TERMINATOR. In Schneider and Brandt [95], pages 151–156.

[59] R. H. Katz and G. Borriello. *Contemporary Logic Design*. Prentice Hall, 2005.

[60] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[61] A. Koprowski. TPA: termination proved automatically. In Pfenning [92], pages 257–266.

[62] D. Kozen. On Kleene algebras and closed semirings. In B. Rovan, editor, *Mathematical Foundations of Computer Science (MFCS'90)*, volume 452 of *Lecture Notes in Computer Science*, pages 26–47. Springer Verlag, 1990.

[63] A. Krauss. *Defining Recursive Functions in Isabelle/HOL*. Part of the Isabelle documentation. http://isabelle.in.tum.de/doc/functions.pdf.

[64] S. Krstić and J. Matthews. Inductive invariants for nested recursion. In D. A. Basin and B. Wolff, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 253–269. Springer Verlag, 2003.

[65] C. S. Lee. Ranking functions for size-change termination. *ACM Transactions on Programming Languages and Systems*, 2008. to appear.

[66] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Principles of Programming Languages (PoPL 2001)*, pages 81–92, 2001.

[67] Z. Manna and R. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48, 1981.

[68] P. Manolios and J. S. Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.

[69] P. Manolios and D. Vroon. Termination analysis with calling context graphs. In T. Ball and R. B. Jones, editors, *Computer Aided Verification (CAV 2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 401–414. Springer Verlag, 2006.

[70] J. Matthews. Recursive function definition over coinductive types. In Bertot et al. [16], pages 73–90.

[71] E. J. McCluskey. Minimization of boolean formulas. *Bell Lab. Tech. J.*, 35(6):1417–1444, Nov 1956.

[72] E. J. McCluskey. *Logic Design Principles.* Prentice Hall, 1986.

[73] J. Meng, L. C. Paulson, and G. Klein. A termination checker for Isabelle Hoare logic. In B. Beckert, editor, *4th International Verification Workshop (VERIFY'07)*, volume 259 of *CEUR Workshop Proceedings*, pages 104–118, 2007.

[74] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML, Revised Edition.* MIT Press, 1997.

[75] Y. Minamide. Verified decision procedures on context-free grammars. In Schneider and Brandt [95], pages 173–188.

[76] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC 2001)*, pages 530–535. ACM, 2001.

[77] O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF=HOL+LCF. *Journal of Functional Programming*, 9(2):191–223, 1999.

[78] O. Müller and K. Slind. Treating partiality in a logic of total functions. *The Computer Journal*, 40(10):640–652, 1997.

[79] M. Muzalewski. An outline of PC mizar. Technical report, Fondation Philippe le Hodey, Brussels, 1993.

[80] W. Naraschewski and M. Wenzel. Object-oriented verification based on record subtyping in higher-order logic. In Grundy and Newey [47], pages 349–366.

[81] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer Verlag, 2002.

[82] T. Nishihara and Y. Minamide. Depth first search. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs.* http://afp.sf.net/entries/Depth-First-Search.shtml, June 2004. Formal proof development.

[83] C. Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, 1999.

[84] S. Owens and K. Slind. Adapting functional programs to higher-order logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, 2008.

[85] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer Verlag, 1992.

[86] P. Pandya and M. Joseph. A structure-directed total correctness proof rule for recursive procedure calls. *The Computer Journal*, 29(6):531–537, 1986.

[87] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, New York, 1994.

[88] L. C. Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5:143–170, 1985.

[89] L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *Automated Deduction (CADE-12)*, volume 814 of *Lecture Notes in Computer Science*, pages 148–161. Springer Verlag, 1994.

[90] L. C. Paulson. *Isabelle — A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science.* Springer Verlag, 1994.

[91] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Principles of Programming Languages (POPL'93)*, pages 71–84, 1993.

[92] F. Pfenning, editor. *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings*, volume 4098 of *Lecture Notes in Computer Science.* Springer Verlag, 2006.

[93] M. Rhiger. Type-safe pattern combinators. *Journal of Functional Programming.* To appear.

[94] M. Schaefer and C. Umans. Completeness in the polynomial-time hierarchy: Part I: A compendium. *SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 33, 2002.

[95] K. Schneider and J. Brandt, editors. *Theorem Proving in Higher Order Logics (TPHOLs 2007)*, volume 4732 of *Lecture Notes in Computer Science.* Springer Verlag, 2007.

[96] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: effective tool support for the working semanticist. In R. Hinze and N. Ramsey, editors, *ICFP*, pages 1–12. ACM, 2007.

[97] K. Slind. Function definition in Higher-Order Logic. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs '96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 381–397. Springer Verlag, 1996.

[98] K. Slind. *Reasoning About Terminating Functional Programs.* PhD thesis, Institut für Informatik, Technische Universität München, 1999.

[99] K. Slind. Another look at nested recursion. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLS 2000)*, volume 1869 of *Lecture Notes in Computer Science*, pages 498–518. Springer Verlag, 2000.

[100] C. Sprenger and D. A. Basin. A monad-based modeling and verification toolbox with application to security protocols. In Schneider and Brandt [95], pages 302–318.

[101] T. Strzemecki. Polynomial-time algorithms for generation of prime implicants. *J. Complexity*, 8(1):37–63, 1992.

[102] Termination competition.
http://termination-portal.org/wiki/Termination_Competition.

[103] R. Thiemann and J. Giesl. Size-change termination for term rewriting. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 264–278. Springer Verlag, 2003.

[104] R. Thiemann and J. Giesl. The size-change principle and dependency pairs for termination of term rewriting. *Appl. Algebra Eng. Commun. Comput.*, 16(4):229–270, 2005.

[105] S. Thompson. *Haskell: The Craft of Functional Programming (2nd Edition)*. Addison-Wesley, 1999.

[106] C. Umans, T. Villa, and A. L. Sangiovanni-Vincentelli. Complexity of two-level logic minimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(7):1230–1246, 2006.

[107] C. M. Umans. *Approximability and completeness in the polynomial hierarchy*. PhD thesis, University of California, Berkeley, 2000.

[108] C. Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

[109] C. Urban and S. Berghofer. A recursion combinator for nominal datatypes implemented in Isabelle/HOL. In Furbach and Shankar [39], pages 498–512.

[110] P. Wadler. Efficient compilation of pattern-matching. In S. L. Peyton Jones, editor, *The Implementation of Functional Programming Languages*, chapter 5. Prentice-Hall International, 1987.

[111] C. Walther. On proving the termination of algorithms by machine. *J. Artificial Intelligence*, 71(1):101–157, 1994.

[112] T. Weber. *SAT-based Finite Model Generation for Higher-Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, Germany, 2008.

[113] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. P. Felty, editors, *Theorem Proving in Higher Order Logics (TPHOLs '97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 307–322. Springer Verlag, 1997.

[114] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, Germany, 2002.

[115] M. Wenzel, L. C. Paulson, and T. Nipkow. The Isabelle framework. In Ait Mohamed et al. [3], pages 33–38.

[116] L. While and T. Field. Optimising parallel pattern-matching by source-level program transformation. In V. Estivill-Castro, editor, *ACSC*, volume 38 of *CRPIT*, pages 239–248. Australian Computer Society, 2005.