Proofs, Programs and Executable Specifications in Higher Order Logic

Stefan Berghofer

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

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

Proofs, Programs and Executable Specifications in Higher Order Logic

Stefan Berghofer

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.

Die Dissertation wurde am 18. Juni 2003 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 7. Oktober 2003 angenommen.

Kurzfassung

Diese Arbeit präsentiert mehrere Erweiterungen des generischen Theorembeweisers Isabelle.

Der zentrale Beitrag der Arbeit ist die Erweiterung von Isabelle um einen Kalkül für primitive Beweisterme, in dem Beweise als Lambda-Terme repräsentiert werden. Primitive Beweisterme erlauben eine unabhängige Verifikation von in Isabelle konstruierten Beweisen durch einen kleinen und vertrauenswürdigen Beweisprüfer und bilden eine wichtige Voraussetzung für den Austausch von Beweisen mit anderen Systemen.

Der Beweistermkalkül wird insbesondere dazu verwendet, um die Beziehung zwischen Beweisen und Programmen zu studieren. Hierzu wird ein Mechanismus zur Extraktion von beweisbar korrekten Programmen aus konstruktiven Beweisen entwickelt und auf verschiedene Fallstudien angewandt.

Darüberhinaus stellen wir einen alternativen Ansatz zur Gewinnung von Programmen aus Spezifikationen vor, der induktive Definitionen direkt als Logikprogramme interpretiert.

Abstract

This thesis presents several extensions to the generic theorem prover Isabelle, a logical framework based on higher order logic.

The central contribution of this thesis is the extension of Isabelle with a calculus of primitive proof terms, in which proofs are represented using λ-terms in the spirit of the Curry-Howard isomorphism. Primitive proof terms allow for an independent verification of proofs constructed in Isabelle by a small and reliable proof checker, and are an important prerequisite for the application of proof transformation and analysis techniques, as well as the exchange of proofs with other systems.

In particular, the proof term calculus is used to study the relationship between proofs and programs. For this purpose, we first develop a generic mechanism for the extraction of provably correct programs from constructive proofs, then instantiate it for the particular object logic Isabelle/HOL, and finally apply it to several case studies, ranging from simple textbook examples to complex applications from the field of combinatorics or the theory of λ-calculus.

Moreover, we introduce an alternative approach for obtaining programs from specifications by directly interpreting inductive definitions as logic programs.

Acknowledgements

First of all, I would like to thank Tobias Nipkow, my supervisor, for introducing me to the field of theorem proving, for offering me a position in his research group and for all his advice and support. I am indebted to Helmut Schwichtenberg for agreeing to act as a referee, for the numerous interesting discussions we had about constructive logic and program extraction, which greatly helped me to shape the ideas presented in this thesis, and for chairing the DFG Graduiertenkolleg "Logic in Computer Science" of which I had the pleasure to be a member.

A big thank-you goes to my former colleague and roommate Markus Wenzel for many inspiring and often controversial discussions, for letting me benefit from his amazing technical knowledge, and for his sometimes unconventional views. Thanks are due to my colleagues Sebastian Skalberg, Martin Strecker, and Tjark Weber for reading draft versions of this thesis and suggesting numerous improvements. I would also like to thank my other current and former colleagues at TU München for the pleasant working atmosphere in our group: Clemens Ballarin, Gertrud Bauer, Gerwin Klein, Farhad Mehta, David von Oheimb, Leonor Prensa-Nieto, Cornelia Pusch, Norbert Schirmer and Martin Wildmoser.

I very much appreciated the lively interaction with my fellow members of the Graduiertenkolleg, which helped to broaden my horizon enormously. In particular, I would like to thank Andreas Abel for discussing with me various questions related to logical frameworks, Laura Crosilla for her comments on parts of this thesis and her encouragement, Felix Joachimski and Ralph Matthes for insightful discussions about λ-calculus and normalization proofs, which inspired me to do a case study on this topic, as well as Monika Seisenberger for a very pleasant and stimulating collaboration, which led to the formalization of Higman's lemma presented in this thesis.

Contents

Chapter 1

Introduction

1.1 Motivation

Interactive theorem provers are tools which allow to build abstract system models, often in some kind of *functional programming language* involving datatypes and recursive functions. They also allow to capture interesting properties of the system to be modelled by a *specification*, which can be expressed in the language of *predicate logic*. Most importantly, they aid in the construction of *proofs*, i.e. formal arguments, showing that a system model satisfies a given specification. This thesis addresses two central issues in theorem proving: The representation of proofs, and the transformation of specifications into executable programs.

Representation of proofs The user interface layer of a theorem prover usually offers rich and expressive *tactic languages* for encoding proof search strategies, or even languages for writing "human-readable" proof documents. During the construction of a proof, the user may also invoke specific decision procedures for fragments of arithmetic, or proof search procedures for propositional or predicate logic, which are built into the theorem prover. Due to this expressiveness, verifying the correctness of such high-level proof descriptions requires a relatively complex machinery, and usually cannot be done independently of the theorem prover. In order to allow for an independent checking of proofs by a relatively small program, a more *primitive* notion of proof is needed.

The insight that *checking* proofs is easier than *finding* proofs already dates back to Aristotle. N.G. de Bruijn, one of the pioneers of computer-assisted theorem proving, was the first to point out that proofs produced by theorem provers should be independently checkable by a small program, in order to achieve a higher confidence in the correctness of the produced results. His Automath system [31] was consequently designed with this principle in mind. Recently, this approach has also been applied to the design of general purpose decision procedures, notably the Cooperative Validity Checker (CVC) developed at Stanford [114]. Instead of simply reporting whether or not a formula is true, CVC also produces proofs for the formulae it claims to be valid. These proofs are formulated in an extension of the Edinburgh Logical Framework [45], and can be verified with a small proof checker called *flea*, which is distributed together with CVC. A recent application of independent proof checking to system security is the *proof-carrying code* methodology by Necula [75, 76]. Proof-carrying code allows for a safe execution of untrusted code, such as device drivers or dynamic web applications, by packaging the code together with a proof that it adheres to a specific *security policy*. This proof can then be checked by a simple program on the client side prior to the execution of the code.

Most standard theorem provers such as Isabelle [92] or the HOL system [42] are based on the so-called *LCF approach*, pioneered by Milner in the early 1970's, and named after the LCF theorem prover [43] for Scott's logic of computable functions in which it was first used. The LCF methodology, which is sometimes advertized as an alternative to de Bruijn's approach of independently checkable proof objects, requires that all functions in a theorem proving system that produce theorems must eventually call functions of a primitive *kernel* implementing the basic inference rules of the logic. Provided that the implementation of these rules is correct, all theorems obtained in this way are guaranteed to be sound. Hence it is often claimed that constructing explicit proof objects for each theorem is unnecessary in such systems. This is only partially true, however. Since even the inference kernels of LCF-style theorem provers are often relatively large, their correctness is difficult to ensure. Being able to verify proofs by a small and independent proof checker helps to minimize the risks. Moreover, a precise notion of proof objects facilitates the exchange of proofs between different theorem proving systems. Finally, proof objects are a prerequisite for proof transformation and analysis or the extraction of computational content from proofs. This activity of revealing additional information inherent in proofs, which is not immediately obvious at first sight, is often summarized by the catchy slogan "proof mining".

The importance of producing proofs that can be checked independently of a theorem prover has also been recognized quite early by the HOL community. As a response to the needs of safety and security conscious users of the HOL system, Cambridge University and SRI initiated the *Foundations of Secure Formal Methods* project, whose main deliverables were the verification of a proof checker for HOL carried out in HOL itself by von Wright [117], as well as an extension of the HOL system with a library for recording proofs by Wong [122]. Although the proof recorder and checker have been tested on small proofs, their performance on larger proofs has never been examined, and they have not found their way into any of the recent releases of the HOL system.

From specifications to executable programs In order for a theorem prover to be useful as a tool for software development, it should ideally support all phases of the development process, ranging from the abstract specification of a system to the generation of executable code. It is therefore not surprising that the design of calculi supporting the formally verified transition from specifications to code has long been an area of active research. This research has led to *algebraic specification languages* accompanied by various notions of *implementation*, as proposed by Sannella, Tarlecki and Wirsing [106, 107], or to *refinement calculi*, such as the one by Back and von Wright [9], to name just a few examples. In this respect, a formalism which is particularly attractive for computer scientists is that of *constructive logic*, since it comes with a built-in notion of *computability* (or executability). Constructive logic rejects the unrestricted usage of the law of *excluded middle*, which means that $P \vee \neg P$ may only be assumed if an *algorithm* for *deciding* the property P has been shown to exist. In contrast to more heavyweight formalisms for reasoning about decidability, such as recursive function theory, constructive logic has the advantage of offering very much the same look and feel as ordinary mathematics.

Proofs as programs One of the central insights of computer science, which connects the two previously mentioned topics, is that *logic* and *computation* are closely interrelated. The fact that theorem proving is similar to programming, and therefore many techniques for the construction and analysis of programs are also applicable to proofs, has first been pointed out by Brouwer, Heyting and Kolmogorov [25, 49, 60], as well as Curry and Howard [51]. Informally, the relationship between proofs and programs can be summarized as follows:

- A proof of $A \longrightarrow B$ is a program that transforms a proof of A into a proof of B
- A proof of $\forall x.\ P\ x$ is a program that transforms an element x into a proof of $P\ x$
- A proof of $\exists x.\ P\ x$ is a pair consisting of an element x and a proof of $P\ x$
- A proof of $A \wedge B$ is a pair consisting of a proof of A and a proof of B
- A proof of $A \vee B$ is either a proof of A or a proof of B, together with a tag telling which alternative has been chosen

In this setting, proof checking simply amounts to type checking, and the elimination of detours in proofs corresponds to β-reduction of functional programs.

It is commonly agreed that assigning types to programs can avoid a substantial amount of programming errors. However, in most programming languages, the fact that a program is well-typed usually does not guarantee its correctness. As an example, consider the typing judgement

$$\mathit{pred} : \mathit{nat} \Rightarrow \mathit{nat}$$

It merely states that *pred* is a function that takes a natural number as an input and again yields a natural number as an output, but does not say anything about the relationship between the input and the output value. Clearly, such a judgement is not particularly informative. A way to improve this situation is to use more expressive type systems, which are powerful enough to capture *specifications* of programs. If we view a proof as a *program*, and the proposition established by it as its *type*, we can rephrase the above judgement to

$$\mathit{pred} : \forall x :: \mathit{nat}.\ x \neq 0 \longrightarrow (\exists y :: \mathit{nat}.\ x = \mathit{Suc}\ y)$$

which says that the function *pred* yields the *predecessor* y of any non-zero natural number x. This approach, which is often summarized by the slogan *"proofs as programs, propositions as types"*, forms the basis for most modern *type theories*, as implemented for example in the proof assistants Coq [12] developed under the direction of G. Huet at INRIA, as well as the Nuprl system [27] by R. Constable from Cornell University. Certainly the most important consequence of the *proofs as programs* approach is the possibility to *extract* provably correct programs from constructive proofs. Since a constructive proof contains a program together with its proof of correctness, program extraction can also be viewed as a somewhat extreme form of proof-carrying code, or rather "code-carrying proof".

Logical frameworks The reuse of existing components and well-established technology is a major issue in software engineering. This has resulted in the creation of libraries and so-called *frameworks* for specific purposes, such as graphical user interfaces or business applications, which help to shorten the development process by factoring out common functionality. Since a theorem prover is essentially a piece of software, the concept of a framework can be applied to its development as well. Instead of reimplementing theorem provers from scratch for every conceivable logic, it is more advantageous to provide generic algorithms and data structures for representing and reasoning within logics and deductive systems once and for all in the form of a *logical framework*.

Examples of logical frameworks are the Elf system by Pfenning [94], which is an implementation of the Edinburgh Logical Framework (LF), a *dependent* type theory proposed by Harper, Honsell and Plotkin [45], as well as the Isabelle system by Paulson and Nipkow [92], which implements simply typed, minimal higher order logic. While logical frameworks allow to represent and reason *within* deductive sytems (so-called *object logics*), *meta-logical frameworks* also support reasoning *about* such systems, e.g. by induction on the structure of derivations. An example for such a meta-logical framework is the Twelf system [99] by Pfenning and Schürmann, which is the successor of the Elf system.

1.2 Contributions

This thesis is concerned with the development of several extensions to the generic theorem prover Isabelle, a logical framework based on simply typed, minimal higher order logic. Apart from studying theoretical foundations, the emphasis in this thesis is on providing a practically usable implementation. Therefore, all the presented concepts and methods have been implemented in Isabelle, and most of them are part of the current release (Isabelle2003). Although the implementation has been done for a particular theorem proving system, we believe that most of the concepts are actually rather generic and are therefore also applicable to a wide range of other provers, such as the HOL system.

A logical framework with proof terms The central contribution of this thesis is a calculus of primitive proof terms for Isabelle, in which proofs are represented using λ-terms in the spirit of the Curry-Howard isomorphism. We present a method for synthesizing proof terms step by step via *higher order resolution*, which is the central principle of proof construction in Isabelle. The representation of proofs as terms allows techniques from the area of type checking and term rewriting to be applied to them quite easily, which is in contrast to proof formats such as the one proposed for HOL by von Wright and Wong [117, 122], where a proof is essentially viewed as a "flat" and unstructured list of inferences.

To make the proof term calculus suitable for practical applications, which usually involve proofs of considerable size, we introduce an extended version of the original proof term calculus, allowing for the omission of redundant information in proofs. This calculus of *partial* proofs is accompanied by an algorithm for the reconstruction of omitted information in proofs, using a constraint solving strategy similar to type inference algorithms in functional programming languages. Based on this calculus, we then develop several algorithms for eliminating syntactic redundancies in proof terms, and analyze their performance. The most powerful of these algorithms achieve a compression ratio of over 90%.

This new calculus for proof terms allows to check proofs constructed in Isabelle by a small proof checker, which leads to a much higher degree of reliability. Interestingly, the addition of proof terms to Isabelle even helped to uncover a rather subtle soundness bug in the kernel, and to spot some inefficiencies in proof procedures. Apart from increasing the reliability of Isabelle, the proof term calculus also opens up new fields of applications, such as proof-carrying code, program extraction or the exchange of proofs with other theorem provers.

Proofs for equational logic Without additional support in the form of automated proof procedures for specific purposes, interactive theorem provers would be rather tedious to use.

One of the central proof methods of every theorem prover is *term rewriting*. We have redesigned the term rewriting algorithm used in Isabelle such that it generates proofs and therefore can be implemented in a safe way outside the trusted kernel of the theorem prover. Moreover, our algorithm improves on previous ones in that it allows contextual rewriting with unlimited mutual simplification of premises.

A generic framework for program extraction To demonstrate that our proof term calculus is suitable for nontrivial applications, we use it to develop a generic framework for extracting programs from constructive proofs. While the correctness of similar program extraction mechanisms to be found in other theorem provers such as Coq is often justified by complex meta-theoretic arguments on paper only [86], which is somewhat unsatisfactory from the viewpoint of machine-checked proofs, our framework also yields a correctness proof for each extracted program, which can be checked *inside* the logic. The program extraction framework is based on the concept of *modified realizability* due to Kreisel and Kleene [59].

Moreover, we present an instantiation of the generic framework to the object logic Isabelle/HOL, which also covers advanced constructs such as inductive datatypes and predicates, for which we introduce specific realizability interpretations. Our implementation of program extraction is the first one for a theorem prover of the HOL family. Although, strictly speaking, HOL is a *classical* logic, our work shows that this has very little impact on program extraction. This confirms an observation already made by Harrison [46, §8], who pointed out that very few theorems in the HOL library are inherently classical, and therefore suggested to introduce classical axioms as late as possible in the development process.

Case studies By means of several case studies, we show the practical applicability of the framework for program extraction, and also give an overview of the techniques necessary for program development using constructive logic. The case studies range from relatively simple examples to fairly complex ones, showing that our framework scales up well to larger applications. Among the examples which we have treated is *Higman's lemma*, an interesting result from the theory of combinatorics, as well as a novel formalization of an elegant and short proof of *weak normalization* for the simply-typed λ-calculus, from which an algorithm for normalizing terms can be extracted.

Specifications as logic programs As an alternative approach to obtaining executable programs from constructive proofs, we examine how specifications involving inductive predicates, which are essentially PROLOG-style *Horn clauses*, can be directly interpreted as logic programs. We present a lightweight mechanism for translating logic programs to functional programs, which is based on an annotation of predicates with possible directions of *dataflow*, so-called *modes*. We give an algorithm for the automatic inference of modes and explain how the mode system can be extended to handle combinations of functions and predicates or higher order concepts.

1.3 Overview

The remaining part of this chapter is concerned with preliminaries, such as notation and styles of proof presentation. The rest of the thesis is structured as follows:

Chapter 2 introduces a logical framework with proof terms, together with algorithms and techniques for proof synthesis, compression, and reconstruction.

Chapter 3 deals with strategies for the construction of proofs in equational logic.

Chapter 4 describes a generic framework for the extraction of programs from proofs, which is based on the calculus introduced in Chapter 2, as well as its instantiation to the object logic Isabelle/HOL.

Chapter 5 presents several case studies demonstrating the applicability of the program extraction framework introduced in Chapter 4.

Chapter 6 discusses an alternative approach for obtaining programs from specifications, based on logic programming techniques.

Chapter 7 summarizes the achievements made in this thesis, and gives directions for future work.

1.4 Preliminaries

This section introduces some general notation, which will be used in the rest of this thesis. We also give a very brief overview of proof presentation in Isabelle, and introduce some very basic constructs of the object logic Isabelle/HOL. A more formal definition of the Isabelle logical framework will be given later on in §2.2.

1.4.1 Notation

Vector notation We use the notation $\overline{a}$ to denote a list $a_1 \ldots a_n$ of elements. To emphasize that a list has a certain length n, we write $\overline{a}_{\langle n \rangle}$. The brackets around n are used to avoid confusion with $\overline{a}_i$, which denotes an element of a *family* of vectors, such as $\overline{a}_1 \ldots \overline{a}_n$. To express that a list $\overline{a}$ contains an element a_i with $a_i = x$, we write $x \in \overline{a}$.

Substitutions We use $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ to denote substitutions on types, terms, and proofs. Application of a substitution is written as $u\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$.

Basic types Type variables are denoted by α, β, ... In concrete Isabelle examples, we will also use the ML-like notation $'a$, $'b$, ... The type of functions from σ to τ is written as $\sigma \Rightarrow \tau$.

Isabelle/HOL Most examples in this thesis will be done in the logic Isabelle/HOL. Apart from the type *bool* of truth values, one of the most central types of HOL is the type α *set* of sets with elements of type α. The fact that x is an element of set S is denoted by $x \in S$, where $\in :: \alpha \Rightarrow \alpha\ \mathit{set} \Rightarrow \mathit{bool}$. Set comprehension, i.e. the set of all elements with a particular property P is written as $\{x.\ P\ x\}$, which is syntactic sugar for *Collect* P, where $\mathit{Collect} :: (\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha\ \mathit{set}$. The distinction between the type of sets α *set* and the type of predicates $\alpha \Rightarrow \mathit{bool}$ is mainly of technical nature, and we will often treat these types as if they were identical. Isabelle/HOL also supports *inductive datatypes*. We will not formally introduce the concept of a datatype here, but refer the reader to §4.3.4. Some of the most important datatypes defined in Isabelle/HOL are:

Natural numbers The type *nat* of natural numbers has the constructors $0 :: nat$ and $Suc :: nat \Rightarrow nat$.

Lists The type α *list* of lists has the constructors $Nil :: \alpha\ list$ and $Cons :: \alpha \Rightarrow \alpha\ list \Rightarrow \alpha\ list$, for which there is the concrete syntax [] and $x\ \#\ xs$, respectively. Concatenation of two lists *xs* and *ys* is denoted by $xs\ @\ ys$.

Products The type $\alpha \times \beta$ of products has the constructor $Pair :: \alpha \Rightarrow \beta \Rightarrow \alpha \times \beta$, with concrete syntax $(x,\ y)$. The destructor for pairs is $split :: (\alpha \Rightarrow \beta \Rightarrow \gamma) \Rightarrow \alpha \times \beta \Rightarrow \gamma$, and $\lambda(x,\ y).\ t$ is syntactic sugar for $split\ (\lambda x\ y.\ t)$.

Sums The type $\alpha + \beta$ of disjoint sums has the constructors $Inl :: \alpha \Rightarrow \alpha + \beta$ and $Inr :: \beta \Rightarrow \alpha + \beta$.

The logical operators and inference rules of Isabelle/HOL will be described when introducing the logical framework in §2.2.2. More information about Isabelle/HOL can be found e.g. in the tutorial by Nipkow, Paulson and Wenzel [83].

1.4.2 Styles of proof presentation

All the proofs in the case studies presented in this thesis have been carried out in Isabelle, and the most important of them will be discussed in more detail. Therefore, this section briefly introduces the formats of proof presentation supported by Isabelle. Essentially, the user can choose between two main styles of describing proofs, namely *tactic scripts* and *readable proof documents*. In this thesis, mainly the latter approach is used.

Like most interactive theorem provers, Isabelle can be operated using a set of *tactic* commands, which allow a step-by-step refinement of proof goals. Tactic-based proofs are usually conducted in a backward manner, i.e. a complex goal, which is used as a starting point, is successively broken down into simpler goals by the application of tactics, until ending up with a collection of goals which are trivial to solve, because they directly follow from axioms or assumptions in the context. The proof of *thm1* shown in Figure 1.1 is an example for such a tactic-based proof. Tactics (or *proof methods*) are applied using the **apply** command, where the *rule* method is used for the application of *introduction rules*, while *erule* denotes application of *elimination rules*. Unsurprisingly, the *assumption* method solves the current goal by choosing a suitable assumption from the goal's current context.

An obvious problem with the proof of *thm1* is that it is almost incomprehensible without executing it step by step in a theorem prover. By just looking at the proof script, it is neither clear which goal has to be proved in a particular step, nor how the current context of assumptions looks like. The *Isar* proof language (where Isar stands for *I*ntelligible *s*emi-*a*utomated *r*easoning) developed by Markus Wenzel [120] addresses these deficiencies. In contrast to tactic scripts, proofs written in Isar are quite close to the style of proofs to be found in usual textbooks about mathematics. Isar proof documents are interpreted using a kind of *virtual machine*. Since this virtual machine only relies on functions offered by the underlying core inference kernel of the theorem prover, this guarantees soundness of the theorems produced as a result of this interpretation process.

The proof of *thm2* shown in Figure 1.1 is a reformulation of the proof of *thm1* using Isar. Proofs in Isar either consist of a single application of a proof method, which has the form **by** $\langle \ldots \rangle$, or a more complex proof block of the form **proof** $\langle \ldots \rangle$ **qed** containing further proofs. Each proof block may be started with the application of a proof method, such as *rule impI*.

```
theorem thm1: (∃x. ∀y. P x y) ⟶ (∀y. ∃x. P x y)
  apply (rule impI)
  apply (rule allI)
  apply (erule exE)
  apply (erule allE)
  apply (rule exI)
  apply assumption
  done

theorem thm2: (∃x. ∀y. P x y) ⟶ (∀y. ∃x. P x y)
proof (rule impI)
  assume H: ∃x. ∀y. P x y
  show ∀y. ∃x. P x y
  proof (rule allI)
    fix y
    from H obtain x where ∀y. P x y by (rule exE)
    then have P x y by (rule allE)
    then show ∃x. P x y by (rule exI)
  qed
qed

theorem thm3: (∃x. ∀y. P x y) ⟶ (∀y. ∃x. P x y)
proof
  assume H: ∃x. ∀y. P x y
  show ∀y. ∃x. P x y
  proof
    fix y
    from H obtain x where ∀y. P x y ..
    hence P x y ..
    thus ∃x. P x y ..
  qed
qed
```

Figure 1.1: Different styles of proof

Contexts of local parameters and assumptions are built up using **fix** and **assumes**, respectively. While forward proofs using tactic scripts often tend to be rather cumbersome, Isar supports such proofs just as well as backward proofs. Before finally solving a pending goal using **show** $\langle\ldots\rangle$, several intermediate statements may be established via **have** $\langle\ldots\rangle$. The **then** command indicates that the current result should be used as an input for the following proof step. This is sometimes referred to as *forward chaining*. The **from** command has a similar effect, with the difference that instead of the current result, a list of named facts is used. The command **obtain** x **where** φ $\langle\ldots\rangle$ introduces a new parameter x representing a *witness* for the statement φ, where $\langle\ldots\rangle$ stands for a proof showing that such a witness actually exists. Usually, this proof just consists of an application of the existential elimination rule.

It should be noted that the proof of *thm2* is unnecessarily verbose. As it happens, the system can figure out itself which rules to apply in most situations. In the proof of *thm3*, we have therefore left out all explicit references to inference rules. The **..** command instructs the system to choose a suitable rule from a global set of inference rules suitable for solving the current goal. Moreover, the commands **then have** and **then show** have been replaced by the shortcuts **hence** and **thus**, respectively.

Of course, this brief introduction is by no means complete. A detailed presentation of all concepts and techniques related to readable Isar proof documents is given by Wenzel [120]. For a more gentle introduction, see e.g. the tutorial on Isabelle/Isar by Nipkow [82].

Chapter 2

Proof terms for higher order logic

2.1 Introduction

Isabelle is a *generic* theorem prover. It is generic in the sense that it does not just support a single logic, which is hard-wired into the system, but rather serves as a platform for the implementation of various different logics. Such platforms for implementing logics are usually referred to as *logical frameworks* [97]. In this context, logics implemented using a logical framework are called *object logics*, whereas the language for describing operators, inference rules and proofs of *object logics* is called the *meta logic*.

This chapter is concerned with the representation of proofs in simply-typed, minimal higher order logic, the meta logic provided by Isabelle. We start by introducing the basic calculus in §2.2, which is based on a representation of proofs as λ-terms. In §2.3, we discuss the representation of proofs conducted using *higher-order resolution*, the central proof construction principle in Isabelle. In §2.4, we develop several algorithms for eliminating syntactic redundancies from proofs. This helps to keep proof terms small, which is an indispensable requirement for practical applications. In order to model proofs with omitted syntactic information, we introduce a calculus of *partial* proofs, together with a constraint based reconstruction algorithm for recovering this information.

Figure 2.1 gives an overview of the core infrastructure for proof terms available in Isabelle, which was developed as a part of this thesis. The grey box in the upper left part of the diagram shows the LCF-style kernel of Isabelle, which performs all operations on theorems. It also has to be used by interpreters for tactic scripts or readable formal proof documents. A theorem is essentially an abstract datatype consisting of several fields. Most importantly, one of these contains the proposition of the theorem, while other fields contain the current list of open hypotheses used in the proof of the theorem, or the signature with respect to which the theorem was proved. In order to add proof terms to the kernel, it suffices to extend the theorem datatype with another field, say prf, which holds the proof term corresponding to the theorem. Whenever the theorem is transformed, this proof term is transformed as well, in order to correctly reflect the inferences necessary to establish the theorem. It should be noted that the kernel works with *partial* proofs internally, from which syntactic redundancies have been omitted, since working with full proofs turned out to be too inefficient for practical applications. The elimination of detours, as well as other transformations on proofs can be expressed using *proof rewrite rules*. Due to the isomorphism between proofs and terms, rewriting on proofs can be carried out by an algorithm similar to the one for terms described in §3. It is often useful to be able to do rewriting even on partial proofs, although some rewrite rules may require

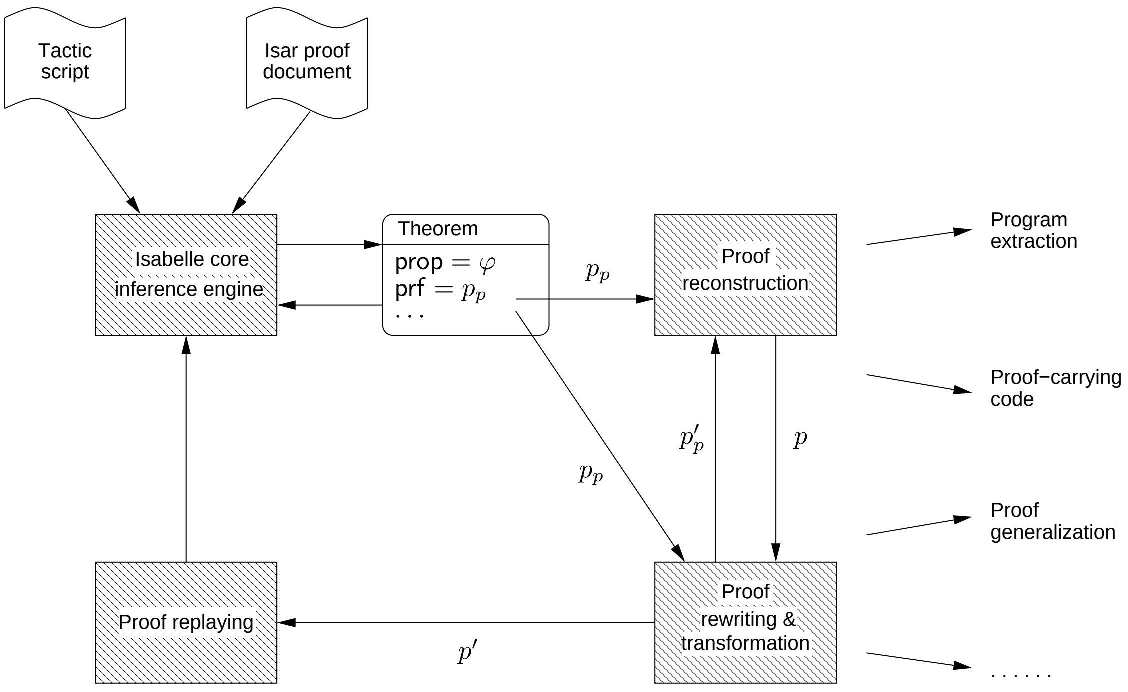


Figure 2.1: Core infrastructure for proof terms

information which is only present in fully reconstructed proofs. A transformed proof may be *replayed* later on, i.e. checked again by executing primitive functions from the inference kernel, which, upon successful completion, again yields a theorem. The program extraction framework described in §4 can be seen as an advanced application for proof transformation and replaying.

2.2 Basic concepts

2.2.1 A logical framework with proofs

Isabelle's meta logic, also called Isabelle/Pure, consists of three layers, which are summarized in Figure 2.2. Isabelle/Pure offers *simple types* according to Church, for which type inference is decidable. The set of type constructors includes the nullary type constructor prop for the type of meta level truth values as well as the binary type constructor $\Rightarrow$ for the function space.

The layer of *terms* is simply-typed λ-calculus, enriched with additional constants, with the usual typing rules. The connectives of the meta logic, namely universal quantification $\bigwedge$ and implication $\Longrightarrow$, are just specific constants, and logical formulae are terms of type prop. The signature Σ is a function mapping each constant to a type, possibly with free type variables. In particular,

$$\begin{array}{lcl} \Sigma(\Longrightarrow) & = & \mathsf{prop} \Rightarrow \mathsf{prop} \Rightarrow \mathsf{prop} \\ \Sigma(\bigwedge) & = & (\alpha \Rightarrow \mathsf{prop}) \Rightarrow \mathsf{prop} \end{array}$$

Isabelle offers *schematic polymorphism* in the style of Hindley and Milner: when referring to a constant c, one may instantiate the type variables occurring in its declared type $\Sigma(c)$. Unlike in more expressive dependent type theories, no explicit abstraction and application is provided for types.

$$\tau, \sigma \; = \; \alpha \mid (\tau_1, \ldots, \tau_n) tc \qquad\qquad \text{where } tc \in \{\mathsf{prop}, \Rightarrow, \ldots\}$$

Types

$$t, u, \varphi, \psi \; = \; x \mid c_{\{\overline{\alpha} \mapsto \overline{\tau}\}} \mid t\ u \mid \lambda x :: \tau.\ t \qquad\qquad \text{where } c \in \{\bigwedge, \Longrightarrow, \ldots\}$$

$$\frac{}{\Gamma, x :: \tau, \Gamma' \vdash x :: \tau} \qquad \frac{\Sigma(c) = \tau}{\Gamma \vdash c_{\{\overline{\alpha} \mapsto \overline{\tau}\}} : \tau\{\overline{\alpha} \mapsto \overline{\tau}\}}$$

$$\frac{\Gamma \vdash t :: \tau \Rightarrow \sigma \quad \Gamma \vdash u :: \tau}{\Gamma \vdash t\ u :: \sigma} \qquad \frac{\Gamma, x :: \tau \vdash t :: \sigma}{\Gamma \vdash \lambda x :: \tau.\ t :: \tau \Rightarrow \sigma}$$

Terms

$$p, q \; = \; h \mid c_{\{\overline{\alpha} \mapsto \overline{\tau}\}} \mid p \cdot t \mid p \bullet q \mid \boldsymbol{\lambda} x :: \tau.\ p \mid \boldsymbol{\lambda} h : \varphi.\ p$$

$$\frac{}{\Gamma, h : t, \Gamma' \vdash h : t} \qquad \frac{\Sigma(c) = \varphi}{\Gamma \vdash c_{\{\overline{\alpha} \mapsto \overline{\tau}\}} : \varphi\{\overline{\alpha} \mapsto \overline{\tau}\}}$$

$$\frac{\Gamma \vdash p : \bigwedge x :: \tau.\ \varphi \quad \Gamma \vdash t :: \tau}{\Gamma \vdash p \cdot t : P\{x \mapsto t\}} \qquad \frac{\Gamma, x :: \tau \vdash p : \varphi}{\Gamma \vdash \boldsymbol{\lambda} x :: \tau.\ p : \bigwedge x :: \tau.\ \varphi}$$

$$\frac{\Gamma \vdash p : \varphi \Longrightarrow \psi \quad \Gamma \vdash q : \varphi}{\Gamma \vdash p \bullet q : \psi} \qquad \frac{\Gamma, h : \varphi \vdash p : \psi \quad \Gamma \vdash \psi : \mathsf{prop}}{\Gamma \vdash \boldsymbol{\lambda} h : \varphi.\ p : \varphi \Longrightarrow \psi}$$

Proofs

Figure 2.2: The Isabelle/Pure logical framework

The layer of *proofs* is built on top of the layers of terms and types. The central idea behind the proof layer is the *Curry-Howard isomorphism*, according to which proofs can be represented as λ-terms. Consequently, the proof layer looks quite similar to the term layer, with the difference that there are two kinds of abstractions and two kinds of applications, corresponding to introduction and elimination of universal quantifiers and implications, respectively. The proof checking rules for $\Longrightarrow$ can be seen as non-dependent variants of the rules for $\bigwedge$. While the abstraction $(\boldsymbol{\lambda} x :: \tau.\ p)$ corresponding to $\bigwedge$ introduction abstracts over a *term variable* x of type τ, the abstraction $(\boldsymbol{\lambda} h : \varphi.\ p)$ corresponding to $\Longrightarrow$ introduction abstracts over a *hypothesis variable* (or proof variable) h standing for a proof of the proposition φ. The formulae φ and ψ in the proof checking rules are terms of type prop. Proof constants c are references to axioms or other theorems that have already been proved. Function Σ maps each proof constant to a term of type prop. Similar to term constants, one may give an instantiation for the free type variables occurring in the proposition corresponding to the proof constant.
The rules for β-reduction on terms and proofs are as usual:

$$(\lambda x :: \tau.\ t)\ u \longmapsto t\{x \mapsto u\} \qquad (\boldsymbol{\lambda} x :: \tau.\ p) \cdot t \longmapsto p\{x \mapsto t\} \qquad (\boldsymbol{\lambda} h : \varphi.\ p) \bullet q \longmapsto p\{h \mapsto q\}$$

We tacitly allow β-reductions to be performed on terms during proof checking, so there is no need to explicitly record such reduction steps, of which there may be numerous, in the proof

Textbook notation	Logical framework notation
$\dfrac{\begin{matrix}[P]\\ \vdots \\ Q\end{matrix}}{P \longrightarrow Q}$	$\bigwedge P\ Q.\quad (\mathsf{Tr}\ P \Longrightarrow \mathsf{Tr}\ Q) \Longrightarrow \mathsf{Tr}\ (P \longrightarrow Q)$
$\dfrac{\begin{matrix}[x]\\ \vdots \\ P\ x\end{matrix}}{\forall x.\ P\ x}$	$\bigwedge P.\quad (\bigwedge x.\ \mathsf{Tr}\ (P\ x)) \Longrightarrow \mathsf{Tr}\ (\forall x.\ P\ x)$

Figure 2.3: Comparison of textbook and logical framework notation

term. This is a restricted form of what Barendregt [10] calls the *Poincaré principle*, which says that certain computations need not be recorded in the proof term. Other theorem provers such as Coq allow even more complex computation steps to be performed implicitly, in particular δ and ι-reduction, which correspond to unfolding of definitions and application of reduction rules for recursion combinators of inductive datatypes, respectively.

Checking of terms and proofs is performed relative to a context Γ, mapping term variables to types and hypothesis variables to propositions. All contexts occurring in the above proof checking rules are assumed to be well-formed. Intuitively, this means that each term variable must be declared in the context before it may be used in a hypothesis $h : \varphi \in \Gamma$, and all φ must be well-typed terms of type prop. More formally, well-formedness of contexts can be expressed by the following judgement $\vdash_{wf}$:

$$\frac{}{\vdash_{wf} []} \qquad \frac{\vdash_{wf} \Gamma \quad \Gamma \vdash \varphi : \mathsf{prop}}{\vdash_{wf} \Gamma, h : \varphi}$$

Often, one also requires that each variable is declared at most once in the context, although in an implementation using de Bruijn indices, this is not so much of an issue.

2.2.2 Formalizing object logics

Object logics are formalized by introducing a new type of object level truth values, say bool, and by declaring their logical operators as new constants. In order to embed logical formulae of the object logic into the meta logic, one also needs to introduce a *coercion function* $\mathsf{Tr} :: \mathsf{bool} \Rightarrow \mathsf{prop}$, which turns object level truth values into meta level truth values. Intuitively, $\mathsf{Tr}\ P$ should be read as "P is true". Meta-level implication $\Longrightarrow$ and meta-level universal quantification $\bigwedge$ allow for an elegant formalization of *assumption contexts*, as well as *variable conditions* of inference rules with quantifiers, such as "x not free in ...", which are often found in textbook presentations of logics. A comparison of the usual textbook notation and logical framework notation for inference rules is shown in Figure 2.3. The representation of logics in Isabelle is based on the *higher-order abstract syntax* approach, which means that variable bindings are formalized using the λ-abstraction of the underlying term calculus. For example, $\forall x.\ P\ x$ is just an abbreviation for $\forall\ (\lambda x.\ P\ x)$. The coercion function Tr, as well as outermost quantifiers binding variables such as P and Q, which we call the *parameters* of an inference rule, are usually omitted for the sake of readability. Figure 2.4 shows the operators and inference rules for the constructive fragment of the object logic Isabelle/HOL. Instead of

Tr	::	$\mathsf{bool} \Rightarrow \mathsf{prop}$
$\mathsf{True},\ \mathsf{False}$	::	bool
$\neg$	::	$\mathsf{bool} \Rightarrow \mathsf{bool}$
$\longrightarrow,\ \wedge,\ \vee$	::	$\mathsf{bool} \Rightarrow \mathsf{bool} \Rightarrow \mathsf{bool}$
$\forall,\ \exists$	::	$(\alpha \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{bool}$
$=$	::	$\alpha \Rightarrow \alpha \Rightarrow \mathsf{bool}$

Logical operators

impI	:	$(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$	*mp*	:	$P \longrightarrow Q \Longrightarrow P \Longrightarrow Q$
allI	:	$(\bigwedge x.\ P\ x) \Longrightarrow \forall x.\ P\ x$	*spec*	:	$\forall x.\ P\ x \Longrightarrow P\ x$
exI	:	$P\ x \Longrightarrow \exists x.\ P\ x$	*exE*	:	$\exists x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q) \Longrightarrow Q$
conjI	:	$P \Longrightarrow Q \Longrightarrow P \wedge Q$	*conjunct₁*	:	$P \wedge Q \Longrightarrow P$
			conjunct₂	:	$P \wedge Q \Longrightarrow Q$
disjI₁	:	$P \Longrightarrow P \vee Q$	*disjE*	:	$P \vee Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
disjI₂	:	$Q \Longrightarrow P \vee Q$			
notI	:	$(P \Longrightarrow \mathsf{False}) \Longrightarrow \neg P$	*notE*	:	$\neg P \Longrightarrow P \Longrightarrow R$
			FalseE	:	$\mathsf{False} \Longrightarrow P$
TrueI	:	True			
refl	:	$x = x$	*subst*	:	$x = y \Longrightarrow P\ x \Longrightarrow P\ y$

Inference rules

Figure 2.4: Constructive fragment of Isabelle/HOL

the rules $\mathit{conjunct}_1$ and $\mathit{conjunct}_2$, as well as *spec* given above, the rules

$$\begin{array}{lll} \mathit{conjE} & : & P \wedge Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R \\ \mathit{allE} & : & \forall x.\ P\ x \Longrightarrow (P\ x \Longrightarrow R) \Longrightarrow R \end{array}$$

are sometimes more convenient to use in backward proofs. Classical logic can be obtained by adding the rule

$$\mathit{ccontr} :\ (\neg P \Longrightarrow \mathsf{False}) \Longrightarrow P$$

It is also useful to specify rewrite rules for eliminating detours in HOL proofs, such as

$$(\mathit{mp} \cdot A \cdot B \bullet (\mathit{impI} \cdot A \cdot B \bullet \mathit{prf})) \longmapsto \mathit{prf}$$

$$(\mathit{impI} \cdot A \cdot B \bullet (\mathit{mp} \cdot A \cdot B \bullet \mathit{prf})) \longmapsto \mathit{prf}$$

$$(\mathit{spec} \cdot P \cdot x \bullet (\mathit{allI} \cdot P \bullet \mathit{prf})) \longmapsto prf \cdot x$$

$$(\mathit{allI} \cdot P \bullet (\boldsymbol{\lambda} x.\ \mathit{spec} \cdot P \cdot x \bullet \mathit{prf})) \longmapsto \mathit{prf}$$

$$(\mathit{exE} \cdot P \cdot Q \bullet (\mathit{exI} \cdot P \cdot x \bullet \mathit{prf}_1) \bullet \mathit{prf}_2) \longmapsto (\mathit{prf}_2 \cdot x \bullet \mathit{prf}_1)$$

$$(\mathit{exE} \cdot P \cdot Q \bullet \mathit{prf} \bullet (\mathit{exI} \cdot P)) \longmapsto \mathit{prf}$$

$$(\mathit{disjE} \cdot P \cdot Q \cdot R \bullet (\mathit{disjI}_1 \cdot P \cdot Q \bullet \mathit{prf}_1) \bullet \mathit{prf}_2 \bullet \mathit{prf}_3) \longmapsto (\mathit{prf}_2 \bullet \mathit{prf}_1)$$

$$(\mathit{disjE} \cdot P \cdot Q \cdot R \bullet (\mathit{disjI}_2 \cdot Q \cdot P \bullet \mathit{prf}_1) \bullet \mathit{prf}_2 \bullet \mathit{prf}_3) \longmapsto (\mathit{prf}_3 \bullet \mathit{prf}_1)$$

$$(\mathit{conjunct}_1 \cdot P \cdot Q \bullet (\mathit{conjI} \cdot P \cdot Q \bullet \mathit{prf}_1 \bullet \mathit{prf}_2)) \longmapsto \mathit{prf}_1$$

$$(\mathit{conjunct}_2 \cdot P \cdot Q \bullet (\mathit{conjI} \cdot P \cdot Q \bullet \mathit{prf}_1 \bullet \mathit{prf}_2)) \longmapsto \mathit{prf}_2$$

2.3 Representing backward resolution proofs

In principle, a kernel implementing just the primitive introduction and elimination rules for $\bigwedge$ and $\Longrightarrow$ shown in Figure 2.2 would already be sufficient for a generic theorem prover. However, it would be quite tedious to conduct proofs using only these rules. Therefore, most theorem provers offer *tactics* for breaking down complex goals into simpler ones, thus allowing to build up a proof of a theorem step by step in a backward manner. The main tool for conducting such backward proofs in Isabelle is *higher-order resolution*, which was first introduced by Paulson [90]. Since it can be implemented using only the primitive rules of the meta logic, resolution is actually a derived rule. However, due to efficiency reasons, it is part of the trusted kernel of Isabelle. To put the kernel of Isabelle on a more firm theoretical basis and to ensure that the proofs it produces can be verified by a checker implementing just the rules from Figure 2.2, this section explains how resolution and related proof principles can be expressed as terms of the primitive proof term calculus.

2.3.1 Encoding the proof steps

What sets Isabelle apart from similar theorem provers, like e.g. the HOL system, is its treatment of proof states. In Isabelle, any proof state is represented by a theorem of the form

$$\psi_1 \Longrightarrow \cdots \Longrightarrow \psi_n \Longrightarrow \varphi$$

where φ is the proposition to be proved and $\psi_1, \ldots, \psi_n$ are the remaining subgoals. Each subgoal is of the form $\bigwedge \overline{x_i}.\ \overline{A_i} \Longrightarrow P_i$, where $\overline{x_i}$ and $\overline{A_i}$ is a context of parameters and local assumptions. This form is sometimes referred to as *Harrop normal form* [120, 69].

Resolution A proof of a proposition φ starts with the trivial theorem $\varphi \Longrightarrow \varphi$ whose proof term is the identity function $\lambda h : \varphi.\ h$. The initial proof state is then refined successively using the resolution rule

$$\begin{array}{c} \dfrac{P_1\ \ldots\ P_m}{C}\ R \\[2ex] \dfrac{P'_1\ \ldots\ P'_i\ \ldots\ P'_{m'}}{C'}\ R' \end{array} \quad \longmapsto \quad \theta\left(\dfrac{P'_1 \quad \ldots \quad P'_{i-1} \quad P_1 \quad \ldots \quad P_m \quad P'_{i+1} \quad \ldots \quad P'_{m'}}{C'}\right)$$

$$\text{where} \quad \theta\ C = \theta\ P'_i$$

until a proof state with no more premises is reached. When refining a proof state having the proof term R' using a rule having the proof term R, the proof term for the resulting proof state can be expressed by

$$\theta\ \left(\boldsymbol{\lambda} \overline{q_{\langle i-1\rangle}}\ \overline{p_{\langle m\rangle}}.\ R' \cdot \overline{q_{\langle i-1\rangle}} \cdot \left(R \cdot \overline{p_{\langle m\rangle}}\right)\right)$$

where θ is a unifier of C and P'_i. The first $i-1$ abstractions are used to skip the first $i-1$ premises of R'. The next m abstractions correspond to the new subgoals introduced by R. Seen from the proofs-as-programs perspective, resolution is simply function composition.

Proof by assumption If the formula P_j in a subgoal $\bigwedge \overline{x_{\langle k\rangle}}.\ \overline{P_{\langle n\rangle}} \Longrightarrow P_j$ of a proof state having the proof term R equals one of the assumptions in $\overline{P_{\langle n\rangle}}$, i.e. $P_j \in \{P_1, \ldots, P_n\} = \overline{P_{\langle n\rangle}}$, this subgoal trivially holds and can therefore be removed from the proof state

$$\frac{Q_1 \; \ldots \; Q_{i-1} \quad \bigwedge \overline{x_{\langle k \rangle}}.\ \overline{P_{\langle n \rangle}} \Longrightarrow P_j \quad Q_{i+1} \; \ldots \; Q_m}{C}\, R \quad \longmapsto \quad \frac{Q_1 \; \ldots \; Q_{i-1} \quad Q_{i+1} \; \ldots \; Q_m}{C}$$

$$\text{where} \quad 1 \leq j \leq n$$

The proof term of the new proof state is obtained by supplying a suitable projection function as an argument to R:

$$\boldsymbol{\lambda} \overline{q_{\langle i-1 \rangle}}.\ R \cdot \overline{q_{\langle i-1 \rangle}} \cdot \left(\boldsymbol{\lambda} \overline{x_{\langle k \rangle}}\ \overline{p_{\langle n \rangle}}.\ p_j\right)$$

Lifting rules into a context Before a subgoal of a proof state can be refined by resolution with a certain rule, the context of both the premises and the conclusion of this rule has to be augmented with additional parameters and assumptions in order to be compatible with the context of the subgoal. This process is called *lifting*. Isabelle distinguishes between two kinds of lifting: lifting over assumptions and lifting over parameters. The former simply adds a list of assumptions $\overline{Q_{\langle n \rangle}}$ to both the premises and the conclusion of a rule:

$$\frac{P_1 \quad \ldots \quad P_m}{C}\, R \quad \longmapsto \quad \frac{\overline{Q_{\langle n \rangle}} \Longrightarrow P_1 \quad \ldots \quad \overline{Q_{\langle n \rangle}} \Longrightarrow P_m}{\overline{Q_{\langle n \rangle}} \Longrightarrow C}$$

The proof term for the lifted rule is

$$\boldsymbol{\lambda} \overline{r_{\langle m \rangle}}\ \overline{q_{\langle n \rangle}}.\ R \cdot \left(\overline{\overline{r_{\langle m \rangle}} \cdot \overline{q_{\langle n \rangle}}}\right)$$

where the first m abstractions correspond to the new premises (with additional assumptions) and the next n abstractions correspond to the additional assumptions.

Lifting over parameters replaces all free variables a_i in a rule $R[\overline{a_{\langle k \rangle}}]$ by new variables a'_i of function type, which are applied to a list of new parameters $\overline{x_{\langle n \rangle}}$. The new parameters are bound by universal quantifiers.

$$\frac{P_1\left[\overline{a_{\langle k \rangle}}\right] \quad \ldots \quad P_m\left[\overline{a_{\langle k \rangle}}\right]}{C\left[\overline{a_{\langle k \rangle}}\right]}\, R\left[\overline{a_{\langle k \rangle}}\right] \quad \longmapsto \quad \frac{\bigwedge \overline{x_{\langle n \rangle}}.\ P_1\left[\overline{a'_{\langle k \rangle}\ \overline{x_{\langle n \rangle}}}\right] \quad \ldots \quad \bigwedge \overline{x_{\langle n \rangle}}.\ P_m\left[\overline{a'_{\langle k \rangle}\ \overline{x_{\langle n \rangle}}}\right]}{\bigwedge \overline{x_{\langle n \rangle}}.\ C\left[\overline{a'_{\langle k \rangle}\ \overline{x_{\langle n \rangle}}}\right]}$$

The proof term for the lifted rule looks similar to the one in the previous case:

$$\boldsymbol{\lambda} \overline{r_{\langle m \rangle}}\ \overline{x_{\langle n \rangle}}.\ R\left[\overline{a'_{\langle k \rangle}\ \overline{x_{\langle n \rangle}}}\right] \cdot \left(\overline{\overline{r_{\langle m \rangle}} \cdot \overline{x_{\langle n \rangle}}}\right)$$

2.3.2 Constructing an example proof

We will now demonstrate how a proof term can be synthesized incrementally while proving a theorem in backward style. A proof term corresponding to a proof state will have the general form

$$\boldsymbol{\lambda}(g_1 : \varphi_1) \ldots (g_n : \varphi_n).\ \ldots\ (g_i\ \overline{x_i}\ \overline{h_i})\ \ldots$$

where the bound variables g_1, ..., g_n stand for proofs of the current subgoals which are still to be found. The $\overline{x_i}$ and $\overline{h_i}$ appearing in the proof term $(g_i\ \overline{x_i}\ \overline{h_i})$ are parameters and assumptions which may be used in the proof of subgoal i. They are bound by abstractions occurring in the context of $(g_i\ \overline{x_i}\ \overline{h_i})$ denoted by $\ldots$ in the above proof term. As an example, the construction of a proof term for the theorem

$$(\exists x.\ \forall y.\ P\ x\ y) \longrightarrow (\forall y.\ \exists x.\ P\ x\ y)$$

will be shown by giving a proof term for each proof state. The parts of the proof terms, which are affected by the application of a rule will be shaded. Initially, the proof state is the trivial theorem:

step 0, remaining subgoal: $(\exists x.\ \forall y.\ P\ x\ y) \longrightarrow (\forall y.\ \exists x.\ P\ x\ y)$

$$\boldsymbol{\lambda} g : ((\exists x.\ \forall y.\ P\ x\ y) \longrightarrow (\forall y.\ \exists x.\ P\ x\ y)).\ g$$

We first apply rule *impI*. Applying a suitable instance of this rule to the trivial initial proof term yields

$$\begin{array}{ll} \boldsymbol{\lambda} g : (\exists x.\ \forall y.\ P\ x\ y) \Longrightarrow (\forall y.\ \exists x.\ P\ x\ y). & \\ \quad (\boldsymbol{\lambda} g' : ((\exists x.\ \forall y.\ P\ x\ y) \longrightarrow (\forall y.\ \exists x.\ P\ x\ y)).\ g') \bullet & \}\ \text{proof term from step 0} \\ \quad (\underbrace{impI \cdot (\exists x.\ \forall y.\ P\ x\ y) \cdot (\forall y.\ \exists x.\ P\ x\ y)}_{\text{instance of } impI} \bullet g) & \end{array}$$

and by $\beta\eta$ reduction of this proof term we obtain

step 1, remaining subgoal: $(\exists x.\ \forall y.\ P\ x\ y) \Longrightarrow (\forall y.\ \exists x.\ P\ x\ y)$

$$impI \cdot (\exists x.\ \forall y.\ P\ x\ y) \cdot (\forall y.\ \exists x.\ P\ x\ y)$$

We now apply *allI* to the above proof state. Before resolving *allI* with the proof state, its context has to be augmented with the assumption $\exists x.\ \forall y.\ P\ x\ y$ of the current goal. The resulting proof term is

$$\begin{array}{l} \boldsymbol{\lambda} g : (\bigwedge y.\ \exists x.\ \forall y.\ P\ x\ y \Longrightarrow \exists x.\ P\ x\ y). \\ \quad impI \cdot (\exists x.\ \forall y.\ P\ x\ y) \cdot (\forall y.\ \exists x.\ P\ x\ y) \bullet \qquad \}\ \text{proof term from step 1} \\ \left.\begin{array}{l} \quad\quad ((\boldsymbol{\lambda} h_2 : (\exists x.\ \forall y.\ P\ x\ y \Longrightarrow \bigwedge y.\ \exists x.\ P\ x\ y). \\ \quad\quad\quad \boldsymbol{\lambda} h_1 : (\exists x.\ \forall y.\ P\ x\ y). \\ \quad\quad\quad\quad allI \cdot (\lambda y.\ \exists x.\ P\ x\ y) \bullet (h_2 \bullet h_1)) \bullet \end{array}\right\}\ \text{lifted instance of } allI \\ \left.\begin{array}{l} \quad\quad\quad\quad\quad (\boldsymbol{\lambda} h_3 : (\exists x.\ \forall y.\ P\ x\ y). \\ \quad\quad\quad\quad\quad\quad \boldsymbol{\lambda} y :: \beta.\ g \cdot y \bullet h_3)) \end{array}\right\}\ \text{rearranging quantifiers} \end{array}$$

Note that the premise of *allI*, which will become the new subgoal of the proof state, has the form $\exists x.\ \forall y.\ P\ x\ y \Longrightarrow \bigwedge y.\ \exists x.\ P\ x\ y$ after having been lifted over the assumption $\exists x.\ \forall y.\ P\ x\ y$. Since Isabelle expects the goal g to be in Harrop normal form, we have to apply an additional function to g, which exchanges the quantifier $\bigwedge y$ and the assumption $\exists x.\ \forall y.\ P\ x\ y$. As before, we apply β reduction to the proof term, which yields

step 2, remaining subgoal: $\bigwedge y.\ \exists x.\ \forall y.\ P\ x\ y \Longrightarrow \exists x.\ P\ x\ y$

$$\begin{array}{l} \boldsymbol{\lambda} g : (\bigwedge y.\ \exists x.\ \forall y.\ P\ x\ y \Longrightarrow \exists x.\ P\ x\ y). \\ \quad impI \cdot (\exists x.\ \forall y.\ P\ x\ y) \cdot (\forall y.\ \exists x.\ P\ x\ y) \bullet \\ \quad\quad (\boldsymbol{\lambda} h_1 : (\exists x.\ \forall y.\ P\ x\ y). \\ \quad\quad\quad allI \cdot (\lambda y.\ \exists x.\ P\ x\ y) \bullet (\boldsymbol{\lambda} y :: \beta.\ g \cdot y \bullet h_1)) \end{array}$$

By eliminating the existential quantifier using *exE* we get *step 3*, remaining subgoal: $\bigwedge y\ x.\ \forall y.\ P\ x\ y \Longrightarrow \exists x.\ P\ x\ y$

$$\begin{array}{l}
\boldsymbol{\lambda} g : (\bigwedge y\ x.\ \forall y.\ P\ x\ y \Longrightarrow \exists x.\ P\ x\ y).\\
\quad \mathit{impI} \cdot (\exists x.\ \forall y.\ P\ x\ y) \cdot (\forall y.\ \exists x.\ P\ x\ y) \bullet\\
\quad\quad (\boldsymbol{\lambda} h_1 : (\exists x.\ \forall y.\ P\ x\ y).\\
\quad\quad\quad \mathit{allI} \cdot (\lambda y.\ \exists x.\ P\ x\ y) \bullet\\
\quad\quad\quad\quad (\boldsymbol{\lambda} y :: \beta.\ \mathit{exE} \cdot (\lambda x.\ \forall y.\ P\ x\ y) \cdot (\exists x.\ P\ x\ y) \bullet h_1 \bullet (\ g \cdot y\ )))
\end{array}$$

Applying the introduction rule *exI* for the existential quantifier results in

step 4, remaining subgoal: $\bigwedge y\ x.\ \forall y.\ P\ x\ y \Longrightarrow P\ (?x\ y\ x)\ y$

$$\begin{array}{l}
\boldsymbol{\lambda} g : (\bigwedge y\ x.\ \forall y.\ P\ x\ y \Longrightarrow P\ (?x\ y\ x)\ y).\\
\quad \mathit{impI} \cdot (\exists x.\ \forall y.\ P\ x\ y) \cdot (\forall y.\ \exists x.\ P\ x\ y) \bullet\\
\quad\quad (\boldsymbol{\lambda} h_1 : (\exists x.\ \forall y.\ P\ x\ y).\\
\quad\quad\quad \mathit{allI} \cdot (\lambda y.\ \exists x.\ P\ x\ y) \bullet\\
\quad\quad\quad\quad (\boldsymbol{\lambda} y :: \beta.\ \mathit{exE} \cdot (\lambda x.\ \forall y.\ P\ x\ y) \cdot (\exists x.\ P\ x\ y) \bullet h_1 \bullet\\
\quad\quad\quad\quad\quad (\boldsymbol{\lambda} x :: \alpha.\\
\quad\quad\quad\quad\quad\quad \boldsymbol{\lambda} h_2 : (\forall y.\ P\ x\ y).\\
\quad\quad\quad\quad\quad\quad\quad \mathit{exI} \cdot (\lambda x.\ P\ x\ y) \cdot (?x\ y\ x) \bullet (\ g \cdot y \cdot x \bullet h_2\ ))))
\end{array}$$

Note that we do not have to give the witness for the existential statement immediately. In place of the witness, a *unification variable* $?x\ y\ x$ is introduced, which may get instantiated later on in the proof. The unification variable is *lifted* over all parameters occurring in the context of the current subgoal.

We now eliminate the universal quantifier using *allE*, which yields

step 5, remaining subgoal: $\bigwedge y\ x.\ P\ x\ (?y\ y\ x) \Longrightarrow P\ (?x\ y\ x)\ y$

$$\begin{array}{l}
\boldsymbol{\lambda} g : (\bigwedge y\ x.\ P\ x\ (?y\ y\ x) \Longrightarrow P\ (?x\ y\ x)\ y).\\
\quad \mathit{impI} \cdot (\exists x.\ \forall y.\ P\ x\ y) \cdot (\forall y.\ \exists x.\ P\ x\ y) \bullet\\
\quad\quad (\boldsymbol{\lambda} h_1 : (\exists x.\ \forall y.\ P\ x\ y).\\
\quad\quad\quad \mathit{allI} \cdot (\lambda y.\ \exists x.\ P\ x\ y) \bullet\\
\quad\quad\quad\quad (\boldsymbol{\lambda} y :: \beta.\ \mathit{exE} \cdot (\lambda x.\ \forall y.\ P\ x\ y) \cdot (\exists x.\ P\ x\ y) \bullet h_1 \bullet\\
\quad\quad\quad\quad\quad (\boldsymbol{\lambda} x :: \alpha.\\
\quad\quad\quad\quad\quad\quad \boldsymbol{\lambda} h_2 : (\forall y.\ P\ x\ y).\\
\quad\quad\quad\quad\quad\quad\quad \mathit{exI} \cdot (\lambda x.\ P\ x\ y) \cdot (?x\ y\ x) \bullet\\
\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{allE} \cdot (P\ x) \cdot (?y\ y\ x) \cdot (P\ (?x\ y\ x)\ y) \bullet h_2 \bullet (\ g \cdot y \cdot x\ )))))
\end{array}$$

Again, no specific term needs to be provided for instantiating the universally quantified variable. Similar to the case of $\exists$-introduction, the quantified variable is replaced by a unification variable $?y\ y\ x$.

We can now prove the remaining subgoal by assumption, which is done by substituting the projection function $\boldsymbol{\lambda}(y :: \beta)\ (x :: \alpha).\ \boldsymbol{\lambda} h_3 : (P\ x\ y).\ h_3$ for g. In order for the goal to be solvable by assumption, we have to solve the unification problem $\{x =^? ?x\ y\ x,\ ?y\ y\ x =^? y\}$, which has the most general unifier[1] $\theta = \{?x \mapsto (\lambda y\ x.\ x),\ ?y \mapsto (\lambda y\ x.\ y)\}$. Thus, the final proof state is

[1]More details of the unification algorithm used in Isabelle are given in §2.4.1

step 6, no subgoals

$$\begin{array}{l}
\mathit{impI} \cdot (\exists x.\ \forall y.\ P\ x\ y) \cdot (\forall y.\ \exists x.\ P\ x\ y) \bullet \\
\quad (\boldsymbol{\lambda} h_1 : (\exists x.\ \forall y.\ P\ x\ y). \\
\quad\quad \mathit{allI} \cdot (\lambda y.\ \exists x.\ P\ x\ y) \bullet \\
\quad\quad\quad (\boldsymbol{\lambda} y :: \beta.\ \mathit{exE} \cdot (\lambda x.\ \forall y.\ P\ x\ y) \cdot (\exists x.\ P\ x\ y) \bullet h_1 \bullet \\
\quad\quad\quad\quad (\boldsymbol{\lambda} x :: \alpha. \\
\quad\quad\quad\quad\quad \boldsymbol{\lambda} h_2 : (\forall y.\ P\ x\ y). \\
\quad\quad\quad\quad\quad\quad \mathit{exI} \cdot (\lambda x.\ P\ x\ y) \cdot x \bullet \\
\quad\quad\quad\quad\quad\quad\quad (\mathit{allE} \cdot (P\ x) \cdot y \cdot (P\ x\ y) \bullet h_2 \bullet (\boldsymbol{\lambda} h_3 : (P\ x\ y).\ h_3)))))
\end{array}$$

When examining the proof objects synthesized for each single step of the proof, in particular for step *2*, which has been explained in more detail above, it should become obvious that the straightforward application of the rules presented in §2.3.1 often leads to proof objects which contain a considerable amount of detours. These detours can be eliminated by *normalizing* the proof term. Although normalization of λ-terms and derivations can be expensive and may also blow up their size, it is usually well-behaved in practice. For example, normalizing the above proof leads to a reduction in size of about 77%. As will be shown in §2.4.2, the fact that a proof term is in normal form is also crucial for the applicability of proof compression and reconstruction algorithms.

Concerning the implementation, it is important to note that no normalization is performed until proof synthesis is completed. This is due to the fact that often not only *proof construction*, but also *proof search* is done using the resolution tactic from the kernel of Isabelle. Therefore, it is important that the rules presented in §2.3.1 can be implemented efficiently. This would not be possible if normalization was applied after each resolution step, since this would each time require a traversal of the whole proof term.

2.4 Partial proof terms

Realistic applications often lead to proof objects of enormous size. For example, Necula [74] reports that his implementation of proof-carrying code based on LF_i produced a security proof with a size of 11 MB for a program which was 2.7 MB in size. Therefore, a proof term calculus must come with a suitable compression technique in order to be practically usable. As a motivating example, consider the following proof of the theorem $A \vee B \longrightarrow B \vee A$:

$$\begin{array}{l}
\mathit{impI} \cdot A \vee B \cdot B \vee A \bullet \\
(\boldsymbol{\lambda} H\!: A \vee B. \\
\quad \mathit{disjE} \cdot A \cdot B \cdot B \vee A \bullet H \bullet (\mathit{disjI2} \cdot A \cdot B) \bullet \\
\quad (\mathit{disjI1} \cdot B \cdot A))
\end{array}$$

Clearly, this proof term contains quite a lot of redundancies: Both $A \vee B$ and $B \vee A$ occur two times, while A and B alone occur even more often. Actually, much of the information in subproofs can be reconstructed from the context they occur in. For example, in the subproof $(\mathit{disjI1} \cdot B \cdot A)$ the term arguments A and B could be reconstructed from the context, which expects this subproof to prove the proposition $B \Longrightarrow B \vee A$. It is the purpose of a compression algorithm to identify and eliminate such redundancies. In the sequel, we will call a proof from which information has been omitted a *partial* (or *implicit*) proof. Due to the Curry-Howard isomorphism, which states that proofs are isomorphic to functional programs, *reconstruction* of omitted information in proofs works quite similar to *type inference* in functional programming languages. Type inference eliminates the need for annotating terms with their types, since one can obtain *constraints* on the required types of terms by examining the context they occur in. Viewing the above proof as an ML program, it would be completely sufficient to write

```
(fn x => case x of disjI1 p => disjI2 p | disjI2 p => disjI1 p);
```

where

```
datatype ('a, 'b) or = disjI1 of 'a | disjI2 of 'b;
```

instead of the much more verbose variant

```
(fn (x : ('a, 'b) or) => case x of
    disjI1 p => (disjI2 : 'a -> ('b, 'a) or) p
  | disjI2 p => (disjI1 : 'b -> ('b, 'a) or) p);
```

with explicit type constraints. Even without explicit type information, any ML compiler can figure out that the given function is type correct and has the most general type `('a, 'b) or -> ('b, 'a) or`.

2.4.1 Reconstruction

This section is concerned with a formal description of the strategy used for reconstructing omitted information in proofs. Since the layer of proofs is built on the layer of terms, this also involves a reconstruction strategy for terms. As mentioned in the introduction, reconstruction works by collecting and solving constraints. The process of collecting constraints is formalized using so-called *reconstruction judgements*, which are defined inductively by a set of inference rules shown in Figure 2.5. The reconstruction judgements for terms and proofs are quite similar in style to the type checking and proof checking judgements presented in Figure 2.2. Omitted information in partial proofs p_p and terms t_p is denoted by placeholders "_". Intuitively, $\Gamma \vdash p_p \rhd (p,\ \varphi,\ C)$ means that the partial proof p_p with placeholders corresponds to a placeholder-free proof p of the proposition φ, provided the constraints C can be satisfied. Constraints, which are usually denoted by the letters C and D, are sets of equations of the form $\tau =^? \sigma$ or $t =^? u$, i.e. equations between types or between terms. We denote by C_t and C_τ the set of all term and type constraints in C, respectively. The positions in partial terms and proofs, in which information may be omitted, are specified by a grammar for t_p and p_p.

- In partial terms, only types may be omitted. Types may appear in two places in Isabelle's term calculus: They may be attached to constants, due to schematic polymorphism, as well as to variables in abstractions.
- In partial proofs, both types and terms may be omitted. In proofs, types may be attached to proof constants, i.e. pre-proved theorems or axioms, as well as to variables in abstractions over term variables, which correspond to $\bigwedge$ introduction. Terms may occur in applications of proofs to terms, which correspond to $\bigwedge$ elimination, and may also be attached to abstractions over hypothesis variables, which correspond to $\Longrightarrow$ introduction.

It does not seem reasonable to allow entire subproofs to be omitted, since this would require actual proof search to be performed during reconstruction. This can make the reconstruction algorithm almost as complex as a theorem prover, which somehow contradicts the design goal of having an independent proof checking and reconstruction algorithm which is as small as possible. We can view this as a trade-off between the size of the proof and the size of the program needed to check or reconstruct the proof.

During reconstruction, placeholders are replaced by *unification variables*, which are prefixed with a question mark, e.g. $?\alpha$ for type variables, or $?f$ for term variables, in order to distinguish them from other variables. Unification variables belong to the set $\mathcal{UV}$, which is assumed to be disjoint from the set $\mathcal{V}$ of ordinary variables. In particular, an abstraction $(\lambda x.\ t)$ may only be formed with $x \in \mathcal{V}$. As usual, all unification variables occurring in the rules presented in Figure 2.5 are assumed to be new. We will sometimes write $?f_\tau$ to emphasize that $?f$ has type τ. For each of the term and proof constructors which may involve placeholders, there are two inference rules: one with placeholders, and one without. On the term layer, the rules $\mathsf{Const_p}$ and $\mathsf{Abs_p}$ introducing type variables for type placeholders correspond to the placeholder-free versions Const and Abs, respectively. On the proof layer, the rules $\mathsf{PConst_p}$ and $\mathsf{AllI_p}$, which introduce type unification variables for type placeholders correspond to PConst and AllI, whereas the rules $\mathsf{ImpI_p}$ and $\mathsf{AllE_p}$ which introduce term unification variables for term placeholders correspond to ImpI and AllE, respectively. Note that an additional premise for checking the well-typedness of terms φ_p or t_p, as in the rules ImpI and AllE, is missing in the rules $\mathsf{ImpI_p}$ and $\mathsf{AllE_p}$. Since a term placeholder may stand for a term depending on any of the term variables in the current context Γ, term unification variables introduced by the rules $\mathsf{ImpI_p}$ and $\mathsf{AllE_p}$ have to be "lifted". For example, $(\boldsymbol{\lambda} h : \_.\ p_p)$ becomes $(\boldsymbol{\lambda} h : ?f_{\overline{\tau_\Gamma} \Rightarrow \mathsf{prop}}\ \overline{V_\Gamma}.\ p)$, where $\overline{V_\Gamma}$ denotes the list of all term variables declared in the context Γ, and $\overline{\tau_\Gamma}$ denotes the list of their types, i.e. $\overline{V_\Gamma} = x_1 \ldots x_n$ and $\overline{\tau_\Gamma} = \tau_1 \ldots \tau_n$ for $\Gamma = x_1 :: \tau_1 \ldots x_n :: \tau_n$. This lifting also applies to term unification variables introduced in a more indirect way by the rules AllE and ImpE.

The reconstruction algorithm on terms, which collects type constraints, is essentially the type inference algorithm which has already been part of Isabelle for quite some time. It is quite similar to Milner's famous *algorithm W* which is used in ML. Strictly speaking, the algorithm actually used in Isabelle is a bit more complex than the one shown in Figure 2.5, due to *order-sorted polymorphism* and *type classes* à la Haskell [80]. Since these concepts do not add to the complexity of proof reconstruction, which is the main subject of this chapter, they have been omitted here.

Due to the Curry-Howard isomorphism, the reconstruction algorithm on proofs, which collects both term and type constraints, can be viewed as a generalization of the reconstruction algorithm on terms. For example, the intuition behind the proof reconstruction rule ImpE, which is the counterpart of the term reconstruction rule App, is as follows: if p proves proposition φ, then φ must be some implication and the proposition ψ proved by q must be the premise of this implication. Moreover, the proposition proved by $(p \cdot q)$ is the conclusion of the implication. The set of constraints for $(p \cdot q)$ is the union of the constraints for p and q, plus one additional constraint expressing that φ is a suitable implication.

Constraints between terms and types generated by the reconstruction judgements are solved using *unification*. Unification computes a substitution θ for unification variables such as $?\alpha$ and $?f$. We assume that $\mathit{Vars}(\theta(?v)) \subseteq \mathcal{UV}$, i.e. the term substituted for a term unification variable $?v$ may not contain any free variables other than unification variables. Since terms contain types, solving a constraint between terms may also yield an instantiation for some of the type variables occurring in the terms.

Definition 2.1 (Solution of constraints) A substitution θ is called a *solution* of the constraint set C iff

- for all $\tau =^? \sigma \in C$, $\theta(\tau) = \theta(\sigma)$, and
- for all $t =^? u \in C$, $\theta(t) =_{\beta\eta} \theta(u)$

The correspondence between type checking and term reconstruction, as well as proof checking and proof reconstruction, can now be stated as follows:

Theorem 2.1 (Soundness of term and proof reconstruction)

- If $\Gamma \vdash t_p \rhd (t,\ \tau,\ C)$ and θ is a solution of C, then $\theta(\Gamma) \vdash \theta(t) :: \theta(\tau)$
- If $\Gamma \vdash p_p \rhd (p,\ \varphi,\ C)$ and θ is a solution of C, then $\theta(\Gamma) \vdash \theta(p) : \theta(\varphi)$

Proof: by induction on the derivations of $\Gamma \vdash t_p \rhd (t,\ \tau,\ C)$ and $\Gamma \vdash p_p \rhd (p,\ \varphi,\ C)$.

Type variables in Isabelle are *first order* in the sense that they may not range over *type constructors*. Therefore, *first order unification* suffices to solve constraints between types, which is decidable and yields most general solutions. In particular, it is decidable for a given term t_p with $\Gamma \vdash t_p \rhd (t,\ \tau,\ C)$, whether a (most general) solution of C exists. Thus, we may safely omit any typing information from a term, without compromising decidability of type inference.

Unfortunately, the same cannot be said for constraints between terms. Since term unification variables may also range over functions, solving constraints between terms requires *higher order unification*, which has been shown to be undecidable in general by Huet [39]. Miller [69] has identified a fragment of higher order terms, so-called *higher order patterns*, for which unification is decidable and yields most general solutions.

Definition 2.2 (Higher order pattern) A term t is a *higher order pattern*, $t \in \mathcal{P}at$ for short, if all occurrences of unification variables $?v$ in t are of the form $(?v\ x_1 \ldots x_n)$, where $x_1 \ldots x_n$ are *distinct bound variables.*

For example, $(\lambda x\ y.\ ?v\ y\ x)$ is a higher order pattern, whereas $(?v\ ?w)$ and $(\lambda x.\ ?v\ x\ x)$ are not. To see why the above restriction is necessary to guarantee the existence of a most general solution, consider the constraints $(?v\ ?w) =^? ?w$ and $(\lambda x.\ ?v\ x\ x) =^? (\lambda x.\ ?w\ x)$. The former has the solutions $\theta_1 = \{?v \mapsto (\lambda x.\ x)\}$ and $\theta_2 = \{?v \mapsto (\lambda x.\ ?w)\}$, while the latter has the solutions $\theta_1 = \{?v \mapsto (\lambda x\ y.\ ?w\ x)\}$ and $\theta_2 = \{?v \mapsto (\lambda x\ y.\ ?w\ y)\}$. The process of solving constraints is usually specified as a set of transformation rules. The following definition closely follows the one given by Nipkow [79]. A similar description of unification for the more expressive Calculus of Constructions is given by Pfenning [95].

Definition 2.3 (Solvability using pattern unification) Let the relation $\longrightarrow_{\mathcal{S}}$ for solving a *single* constraint be characterized by the rules[2]

$$(\{(\lambda x.\ t) =^? (\lambda x.\ u)\} \uplus C,\ \theta) \longrightarrow_{\mathcal{S}} (\{t =^? u\} \cup C,\ \theta)$$

$$(\{a\ \overline{t} =^? a\ \overline{u}\} \uplus C,\ \theta) \longrightarrow_{\mathcal{S}} (\{t_1 =^? u_1, \ldots, t_n =^? u_n\} \cup C,\ \theta) \qquad \text{where } a \notin \mathcal{UV}$$

$$(\{?v\ \overline{x} =^? ?v\ \overline{y}\} \uplus C,\ \theta) \longrightarrow_{\mathcal{S}} \langle C \mid \{?v \mapsto (\lambda\overline{x}.\ ?w\ \{x_i \mid x_i = y_i\})\} \circ \theta\rangle$$

$$(\{?v\ \overline{x} =^? ?w\ \overline{y}\} \uplus C,\ \theta) \longrightarrow_{\mathcal{S}} \langle C \mid \{?v \mapsto (\lambda\overline{x}.\ ?u\ \overline{x} \cap \overline{y}),\ ?w \mapsto (\lambda\overline{y}.\ ?u\ \overline{x} \cap \overline{y})\} \circ \theta\rangle \quad \text{where } ?v \neq ?w$$

$$(\{?v\ \overline{x} =^? a\ \overline{t}\} \uplus C,\ \theta) \longrightarrow_{\mathcal{S}} \langle C \cup \{?w_1\ \overline{x} =^? t_1, \ldots, ?w_n\ \overline{x} =^? t_n\} \mid \{?v \mapsto \lambda\overline{x}.\ a\ (\overline{?w\ \overline{x}})\} \circ \theta\rangle$$
$$\text{where } a \in \overline{x} \text{ or } a \in \mathcal{C} \text{ and } ?v \notin \mathit{Vars}(\overline{t})$$

where $\mathcal{UV}$ and $\mathcal{C}$ denote the sets of unification variables and constants, respectively, $\overline{x}$ and $\overline{y}$ denote lists[3] of *distinct bound* variables, and

$$\langle C \mid \theta\rangle = (\downarrow_\beta (\theta(C)),\ \theta)$$

Then C is called *solvable using pattern unification* iff $(C,\ \{\}) \longrightarrow^*_{\mathcal{S}} (\{\},\ \theta)$

[2]Note that we ignore constraints on types for the moment.

[3]In situations where the order of arguments is immaterial, we sometimes use sets in place of lists.

$$t_p, u_p, \varphi_p, \psi_p \;=\; x \mid c_{\{\overline{\alpha}\mapsto\overline{\tau}\}} \mid t_p\ u_p \mid \lambda x :: \tau.\ t_p \mid c_{\{\overline{\alpha}\mapsto\_\}} \mid \lambda x :: \_.\ t_p$$

$$\frac{}{\Gamma', x :: \tau, \Gamma \vdash x \rhd (x,\ \tau,\ \{\})}\ \mathsf{Var} \qquad \frac{\Sigma(c) = \tau}{\Gamma \vdash c_{\{\overline{\alpha}\mapsto\overline{\tau}\}} \rhd (c_{\{\overline{\alpha}\mapsto\overline{\tau}\}},\ \tau\{\overline{\alpha}\mapsto\overline{\tau}\},\ \{\})}\ \mathsf{Const}$$

$$\frac{\Gamma, x :: \tau \vdash t_p \rhd (t,\ \sigma,\ C)}{\Gamma \vdash (\lambda x :: \tau.\ t_p) \rhd ((\lambda x :: \tau.\ t),\ \tau \Rightarrow \sigma,\ C)}\ \mathsf{Abs}$$

$$\frac{\Gamma \vdash t_p \rhd (t,\ \varrho,\ C) \quad \Gamma \vdash u_p \rhd (u,\ \tau,\ D)}{\Gamma \vdash (t_p\ u_p) \rhd ((t\ u),\ ?\sigma,\ \{\varrho =^? \tau \Rightarrow ?\sigma\} \cup C \cup D)}\ \mathsf{App}$$

$$\frac{\Sigma(c) = \tau}{\Gamma \vdash c_{\{\overline{\alpha}\mapsto\_\}} \rhd (c_{\{\overline{\alpha}\mapsto\overline{?\alpha}\}},\ \tau\{\overline{\alpha}\mapsto\overline{?\alpha}\},\ \{\})}\ \mathsf{Const_p}$$

$$\frac{\Gamma, x :: ?\alpha \vdash t_p \rhd (t,\ \sigma,\ C)}{\Gamma \vdash (\lambda x :: \_.\ t_p) \rhd ((\lambda x :: ?\alpha.\ t),\ ?\alpha \Rightarrow \sigma,\ C)}\ \mathsf{Abs_p}$$

Terms

$$p_p, q_p \;=\; h \mid c_{\{\overline{\alpha}\mapsto\overline{\tau}\}} \mid p_p \cdot t_p \mid p_p \bullet q_p \mid \boldsymbol{\lambda} x :: \tau.\ p_p \mid \boldsymbol{\lambda} h : \varphi_p.\ p_p \mid c_{\{\overline{\alpha}\mapsto\_\}} \mid p_p \cdot \_ \mid \boldsymbol{\lambda} x :: \_.\ p_p \mid \boldsymbol{\lambda} h : \_.\ p_p$$

$$\frac{}{\Gamma', h : \varphi, \Gamma \vdash h \rhd (h,\ \varphi,\ \{\})}\ \mathsf{Hyp} \qquad \frac{\Sigma(c) = \varphi}{\Gamma \vdash c_{\{\overline{\alpha}\mapsto\overline{\tau}\}} \rhd (c_{\{\overline{\tau}/\overline{\alpha}\}},\ \varphi\{\overline{\tau}/\overline{\alpha}\},\ \{\})}\ \mathsf{PConst}$$

$$\frac{\Gamma \vdash \varphi_p \rhd (\varphi,\ \tau,\ C) \quad \Gamma, h : \varphi \vdash p_p \rhd (p,\ \psi,\ D)}{\Gamma \vdash (\boldsymbol{\lambda} h : \varphi_p.\ p_p) \rhd ((\boldsymbol{\lambda} h : \varphi.\ p),\ \varphi \Longrightarrow \psi,\ C \cup D \cup \{\tau =^? \mathsf{prop}\})}\ \mathsf{ImpI}$$

$$\frac{\Gamma, x :: \tau \vdash p_p \rhd (p,\ \varphi,\ C)}{\Gamma \vdash (\boldsymbol{\lambda} x : \tau.\ p_p) \rhd ((\boldsymbol{\lambda} x : \tau.\ p),\ \bigwedge x :: \tau.\ \varphi,\ \{\lambda x :: \tau.\ r =^? \lambda x :: \tau.\ s \mid (r =^? s) \in C_t\} \cup C_\tau)}\ \mathsf{AllI}$$

$$\frac{\Gamma \vdash p_p \rhd (p,\ \varphi,\ C) \quad \Gamma \vdash q_p \rhd (q,\ \psi,\ D)}{\Gamma \vdash (p_p \bullet q_p) \rhd ((p \bullet q),\ ?f_{\overline{\tau_\Gamma}\Rightarrow\mathsf{prop}}\ \overline{V_\Gamma},\ \{\varphi =^? (\psi \Longrightarrow ?f_{\overline{\tau_\Gamma}\Rightarrow\mathsf{prop}}\ \overline{V_\Gamma})\} \cup C \cup D)}\ \mathsf{ImpE}$$

$$\frac{\Gamma \vdash p_p \rhd (p,\ \varphi,\ C) \quad \Gamma \vdash t_p \rhd (t,\ \tau,\ D)}{\Gamma \vdash (p_p \cdot t_p) \rhd ((p \cdot t),\ ?f_{\overline{\tau_\Gamma},\tau\Rightarrow\mathsf{prop}}\ \overline{V_\Gamma}\ t,\ \{\varphi =^? \bigwedge x :: \tau.\ ?f_{\overline{\tau_\Gamma},\tau\Rightarrow\mathsf{prop}}\ \overline{V_\Gamma}\ x\} \cup C \cup D)}\ \mathsf{AllE}$$

$$\frac{\Sigma(c) = \varphi}{\Gamma \vdash c_{\{\overline{\alpha}\mapsto\_\}} \rhd (c_{\{\overline{\alpha}\mapsto\overline{?\alpha}\}},\ \varphi\{\overline{\alpha}\mapsto\overline{?\alpha}\},\ \{\})}\ \mathsf{PConst_p}$$

$$\frac{\Gamma, h : ?f_{\overline{\tau_\Gamma}\Rightarrow\mathsf{prop}}\ \overline{V_\Gamma} \vdash p_p \rhd (p,\ \psi,\ C)}{\Gamma \vdash (\boldsymbol{\lambda} h : \_.\ p_p) \rhd ((\boldsymbol{\lambda} h : ?f_{\overline{\tau_\Gamma}\Rightarrow\mathsf{prop}}\ \overline{V_\Gamma}.\ p),\ ?f_{\overline{\tau_\Gamma}\Rightarrow\mathsf{prop}}\ \overline{V_\Gamma} \Longrightarrow \psi,\ C)}\ \mathsf{ImpI_p}$$

$$\frac{\Gamma, x :: ?\alpha \vdash p_p \rhd (p,\ \varphi,\ C)}{\Gamma \vdash (\boldsymbol{\lambda} x :: \_.\ p_p) \rhd ((\boldsymbol{\lambda} x :: ?\alpha.\ p),\ \bigwedge x :: ?\alpha.\ \varphi,\ \{\lambda x :: ?\alpha.\ r =^? \lambda x :: ?\alpha.\ s \mid (r =^? s) \in C_t\} \cup C_\tau)}\ \mathsf{AllI_p}$$

$$\frac{\Gamma \vdash p_p \rhd (p,\ \varphi,\ C)}{\Gamma \vdash (p_p \cdot \_) \rhd ((p \cdot ?g_{\overline{\tau_\Gamma}\Rightarrow ?\alpha}\ \overline{V_\Gamma}),\ ?f_{\overline{\tau_\Gamma},\tau\Rightarrow\mathsf{prop}}\ \overline{V_\Gamma}\ (?g_{\overline{\tau_\Gamma}\Rightarrow ?\alpha}\ \overline{V_\Gamma}),\ \{\varphi =^? \bigwedge x :: ?\alpha.\ ?f_{\overline{\tau_\Gamma},\tau\Rightarrow\mathsf{prop}}\ \overline{V_\Gamma}\ x\} \cup C)}\ \mathsf{AllE_p}$$

Proofs

Figure 2.5: Reconstruction judgements

Note that in order for a set of constraints C between terms to be solvable by pattern unification, not all terms in C necessarily need to be patterns. For example,

$$C = \{(\lambda x.\ ?v\ x\ ?w) =^? (\lambda x.\ f\ a\ x),\ (\lambda x\ y.\ f\ x\ y) =^? (\lambda x\ y.\ ?v\ y\ x)\}$$

is solvable using pattern unification, since

$$\begin{array}{l}(C,\ \{\}) \longrightarrow_{\mathcal{S}} (\{(\lambda x.\ f\ ?w\ x) =^? (\lambda x.\ f\ a\ x)\},\ \{?v \mapsto (\lambda y\ x.\ f\ x\ y)\}) \longrightarrow_{\mathcal{S}} \\ (\{\},\ \{?v \mapsto (\lambda y\ x.\ f\ x\ y),\ ?w \mapsto a\})\end{array}$$

although $(\lambda x.\ ?v\ x\ ?w)$ is not a pattern. Note that the *functional* unification algorithm described in [79] is more restrictive than the transformation-based version described above: since it insists on solving constraints in a left-to-right order, it would refuse to solve the unification problem

$$c\ (\lambda x.\ ?v\ x\ ?w)\ (\lambda x\ y.\ f\ x\ y) =^? c\ (\lambda x.\ f\ a\ x)\ (\lambda x\ y.\ ?v\ y\ x)$$

although, as shown above, it could be solved using the transformation-based approach. It should also be noted that if C is large, searching for a solvable constraint $(t =^? u)$ and applying a substitution θ to all constraints in C can be expensive. Therefore, in the implementation of the calculus presented in Figure 2.5, it has turned out to be more advantageous not to collect all constraints first and then try to solve them, but to immediately solve constraints "on the fly" which are in the pattern fragment, and accumulate only those in the constraint set C which are not immediately solvable.

2.4.2 Compression

The aim of this section is to develop compression strategies for omitting as much information from proofs as possible, while still being able to reconstruct the omitted information, i.e. solve the constraints generated by the reconstruction judgement using only pattern unification. More formally, we will focus on the question of how to obtain from a proof p with $\Gamma \vdash p : \varphi$ a partial proof p_p with $\Gamma \vdash p_p \rhd (p',\ \varphi',\ C)$ such that $C \cup \{\varphi' =^? \varphi\}$ is solvable using pattern unification. As noted in the previous section, all typing information can safely be omitted, since type variables are *first order* and their reconstruction is decidable. Thus, the main issue to be dealt with is the omission of terms.

Roughly speaking, compression algorithms for proof terms fall into two categories:

Static algorithms determine redundant term arguments of a proof constant c solely by using information from the signature Σ, i.e. by examining the structure of the proposition $\Sigma(c)$ corresponding to c.

Dynamic algorithms determine the redundant term arguments by considering not only information from the signature Σ, but also from the context a proof of the form $(c \cdot t_1 \cdot \cdots \cdot t_n)$ occurs in. This essentially amounts to performing a kind of *dataflow analysis* on the proof.

Other authors, e.g. Necula [77], use a similar distinction between *local* and *global* algorithms. While dynamic algorithms may sometimes be able to omit more information than static algorithms, they are potentially slower. Moreover, a static compression scheme may also be used effectively to already decrease the size of a proof while it is synthesized as described in §2.3. This is of particular importance for the practical usability of the infrastructure for proof terms: The larger the proofs are which arise during synthesis, the slower proof synthesis will be, since some of the operations performed during proof synthesis require a traversal of the whole proof term. A dynamic algorithm cannot be used effectively on such "incomplete" proofs, since further synthesis steps may introduce additional context information into the proof, which may allow for more term arguments to be omitted. Thus, a dynamic algorithm would have to be run again and again during proof synthesis, e.g. when the proof size exceeds a certain limit, each time performing a traversal of the whole proof term. This could slow down proof synthesis considerably.

Before starting with the discussion of different compression techniques, we list some design considerations which are common to all of the approaches:

1. In an abstraction $(\boldsymbol{\lambda} h : \varphi.\ p)$ over a hypothesis variable h, the term annotation φ will always be omitted by the compression algorithm.
2. Term arguments of *proof constants* c, i.e. some of the terms t_i in a proof of the form $(c \cdot t_1 \cdot \cdots \cdot t_n)$ will be omitted by the compression algorithm under specific conditions.
3. Unlike term arguments of proof constants, term arguments of *hypothesis variables* will never be omitted by the compression algorithm, i.e. in a proof of the form $(h \cdot t_1 \cdot \cdots \cdot t_n)$, none of the t_i will be omitted.

The first of these design decisions is based on the observation that it is still possible to reconstruct a proof if all term annotations in abstractions over hypothesis variables are omitted, provided the proof is in *normal form*, contains no other placeholders and we already know the proposition it is supposed to prove.

Theorem 2.2 (Reconstruction of proofs in normal form) Let p_p be a proof in normal form such that

1. all abstractions over hypothesis variables are of the form $(\boldsymbol{\lambda} h : \_.\ q_p)$, and
2. for all applications $(q_p \cdot t_p)$ we have $t_p \neq \_$ and t_p is ground.

Moreover, let Γ be a context and θ be a substitution such that $\theta(\varphi_i)$ is ground for all $(h_i : \varphi_i) \in \Gamma$. Finally, let $\Gamma \vdash p_p \rhd (p,\ \varphi,\ C)$ and ψ be ground. Then the constraints $C \cup \{\varphi =^? \psi\}$ are either solvable using pattern unification, i.e.

$$\langle C \cup \{\varphi =^? \psi\} \mid \theta\rangle \longrightarrow_{\mathcal{S}}^{*} (\{\},\ \theta')$$

or no solution exists.

Proof: By induction on the size of p_p. There are essentially two cases to consider:

- p_p is an *abstraction*, i.e. $p_p = (\boldsymbol{\lambda} h : \_.\ q_p)$ or $p_p = (\boldsymbol{\lambda} x :: \_.\ q_p)$. We just consider the former case, where $\Gamma, h : ?f\ \overline{V_\Gamma} \vdash q_p \rhd (q,\ \varphi',\ C)$ and $\varphi = ?f\ \overline{V_\Gamma} \Longrightarrow \varphi'$. Now assume that $\psi = \psi'' \Longrightarrow \psi'$ for ψ'' and ψ' ground. Note that

  $$\begin{array}{l}\langle C \cup \{\varphi =^? \psi\} \mid \theta\rangle \longrightarrow_{\mathcal{S}} \langle C \cup \{\varphi' =^? \psi',\ ?f\ \overline{V_\Gamma} =^? \psi''\} \mid \theta\rangle \longrightarrow_{\mathcal{S}} \\ \langle C \cup \{\varphi' =^? \psi'\} \mid \{?f \mapsto (\lambda \overline{V_\Gamma}.\ \psi'')\} \circ \theta\rangle\end{array}$$

  Since q_p is smaller than p_p, we have by induction hypothesis that either

  $$\langle C \cup \{\varphi' =^? \psi'\}) \mid \{?f \mapsto (\lambda \overline{V_\Gamma}.\ \psi'')\} \circ \theta\rangle \longrightarrow_{\mathcal{S}}^{*} (\{\},\ \theta')$$

  or no solution exists. Hence, also $\langle C \cup \{\varphi =^? \psi\} \mid \theta\rangle \longrightarrow_{\mathcal{S}}^{*} (\{\},\ \theta')$ or no solution of $C \cup \{\varphi =^? \psi\}$ exists. If ψ is not of the form $\psi'' \Longrightarrow \psi'$, the constraints are unsolvable.

- p_p is an *application.* Since p_p is in *normal form*, the head of p_p must either be a hypothesis variable or a constant, i.e. $p_p = h_i \cdot \overline{a_p}$ or $p_p = c \cdot \overline{a_p}$. In the sequel, we just consider the former case. By side induction on the length of the argument list $\overline{a_p}$, we prove that if $\Gamma \vdash h_i \cdot \overline{a_p} \rhd (h_i \cdot \overline{a},\ \varphi,\ C)$, then

$$\langle C \mid \theta\rangle \longrightarrow^*_{\mathcal{S}} (\{\},\ \theta') \qquad \text{and} \qquad \theta'(\varphi) \text{ is ground}$$

or no solution exists. As a consequence, we then also get that either

$$\langle C \cup \{\varphi =^? \psi\} \mid \theta\rangle \longrightarrow^*_{\mathcal{S}} \langle \{\varphi =^? \psi\} \mid \theta'\rangle \longrightarrow_{\mathcal{S}} (\{\},\ \theta'')$$

or no solution exists.

  - If $\overline{a}$ is empty, we have $p_p = h_i$ and hence $\Gamma \vdash h_i \rhd (h_i,\ \varphi_i,\ \{\})$, for $(h_i : \varphi_i) \in \Gamma$. Since $C = \{\}$ we have $\theta' = \theta$ and therefore $\theta'(\varphi_i)$ is ground by assumption.
  - In the step case, we have either $p_p = (h_i \cdot \overline{a_p}) \cdot q_p$ or $p_p = (h_i \cdot \overline{a_p}) \cdot t_p$. In the former case, we have $\Gamma \vdash h_i \cdot \overline{a_p} \rhd (h_i \cdot \overline{a},\ \varphi_1,\ C_1)$ and $\Gamma \vdash q_p \rhd (q,\ \varphi_2,\ C_2)$, as well as $\varphi = ?f\ \overline{V_\Gamma}$. By side induction hypothesis, we have that $\langle C_1,\ \mid \theta\rangle \longrightarrow^*_{\mathcal{S}} (\{\},\ \theta')$ and $\theta'(\varphi_1)$ is ground, or no solution exists. Now assume that $\theta'(\varphi_1) = \varphi_2' \Longrightarrow \varphi_3$ for φ_2' and φ_3 ground. If $\theta'(\varphi_1)$ is not of this form, no solution exists. Note that

$$\langle C_1 \cup \{\varphi_1 =^? \varphi_2 \Longrightarrow ?f\ \overline{V_\Gamma}\} \cup C_2 \mid \theta\rangle \longrightarrow^*_{\mathcal{S}} \langle \{\varphi_1 =^? \varphi_2 \Longrightarrow ?f\ \overline{V_\Gamma}\} \cup C_2 \mid \theta'\rangle \longrightarrow_{\mathcal{S}}$$
$$\langle \{\varphi_2' =^? \varphi_2,\ \varphi_3 =^? ?f\ \overline{V_\Gamma}\} \cup C_2 \mid \theta'\rangle \longrightarrow_{\mathcal{S}} \langle \{\varphi_2' =^? \varphi_2\} \cup C_2 \mid \{?f \mapsto (\lambda\overline{V_\Gamma}.\ \varphi_3)\} \circ \theta'\rangle$$

  Since q_p is smaller than p_p and φ_2' is ground, we have by main induction hypothesis that either

$$\langle \{\varphi_2' =^? \varphi_2\} \cup C_2 \mid \{?f \mapsto (\lambda\overline{V_\Gamma}.\ \varphi_3)\} \circ \theta'\rangle \longrightarrow^*_{\mathcal{S}} (\{\},\ \theta'')$$

  or no solution exists. Summing up, we have either

$$\langle C_1 \cup \{\varphi_1 =^? \varphi_2 \Longrightarrow ?f\ \overline{V_\Gamma}\} \cup C_2 \mid \theta\rangle \longrightarrow^*_{\mathcal{S}} (\{\},\ \theta'')$$

  and $\theta''(?f\ \overline{V_\Gamma})$ is ground, or no solution exists.
  The case where $p_p = (h_i \cdot \overline{a_p}) \cdot t_p$ is similar, with the difference that $\Gamma \vdash t_p \rhd (t,\ \tau,\ C_2)$ and $\varphi = ?f\ \overline{V_\Gamma}\ t$. Note that this time, C_2 contains only type constraints, which are solvable using first order unification. By side induction hypothesis, we either have

$$\langle C_1 \cup \{\varphi_1 =^? (\bigwedge x.\ ?f\ \overline{V_\Gamma}\ x)\} \cup C_2 \mid \theta\rangle \longrightarrow^*_{\mathcal{S}} \langle \{\varphi_1 =^? (\bigwedge x.\ ?f\ \overline{V_\Gamma}\ x)\} \cup C_2 \mid \theta'\rangle \longrightarrow^*_{\mathcal{S}}$$
$$\langle C_2 \mid \theta'\rangle \longrightarrow^*_{\mathcal{S}} (\{\},\ \theta'')$$

  where both $\theta'(\varphi_1)$ and hence also $\theta''(?f\ \overline{V_\Gamma}\ t)$ are ground[4], or no solution exists.

To see why it is crucial that the proof to be reconstructed is in normal form, consider the following non-normal proof

$$\boldsymbol{\lambda}(h : \_)\ y\ x.\ \underbrace{(\boldsymbol{\lambda}(h_1 : \_)\ (h_2 : \_).\ h_1)}_{p_1} \cdot (h \cdot x \cdot y) \cdot \underbrace{(\boldsymbol{\lambda}(h_3 : \_).\ h_3 \cdot x \cdot x)}_{p_2}$$

of $(\bigwedge x\ y.\ P\ x\ y) \Longrightarrow (\bigwedge y\ x.\ P\ x\ y)$. Assuming that h_3 is a proof of a proposition $?v_1\ y\ x$ which may depend on y and x, the subproof p_2 generates the constraints

$$C = \{(?v_1\ y\ x) =^? (\bigwedge z.\ ?v_2\ y\ x\ z),\ (?v_2\ y\ x\ x) =^? (\bigwedge z.\ ?v_3\ y\ x\ z)\}$$

[4] t may actually contain type unification variables, but this is unproblematic for pattern unification

of which the second is a non-pattern constraint. For p_2, the proposition $?v_1\ y\ x \Longrightarrow ?v_3\ y\ x\ x$ is inferred. If a ground term denoting the proposition of p_2 was known, this could be used to obtain a ground term for $?v_1$, and therefore, by solving the first constraint, also for $?v_2$. This would then turn the second constraint into a pattern constraint, yielding also a ground term for $?v_3$. However, since we cannot infer the proposition of the second argument of p_1 from its "result proposition" $(\bigwedge y\ x.\ P\ x\ y)$, reconstruction gets stuck.

The two main cases in the above proof dealing with *abstractions* and *applications* closely correspond to the concepts of *canonical* and *atomic objects*, which play an important role in the theory of logical frameworks [97, §6, §7]. Although the proof reconstruction judgement presented in Figure 2.5 does not impose a particular direction of dataflow, it is given implicitly in the above proof of theorem 2.2 by the order in which the constraints are solved. During reconstruction of abstractions, the dataflow is essentially bottom-up, i.e. from the root of the proof term to the leaves, whereas during reconstruction of applications, the dataflow is top-down, i.e. from the leaves to the root. This clear direction of dataflow guarantees that in each step of the constraint solution process, we can always find a constraint $(\varphi =^? \psi) \in C$ such that φ is a pattern and ψ is ground. Since ψ is ground, in this case even *matching* instead of unification would suffice. It is interesting to note that the idea behind the above proof also forms the basis of so-called *bidirectional* or *local* type checking and type inference algorithms. In this context, "local" means that these algorithms only use (typing) information from adjacent nodes in the syntax tree and therefore in principle eliminate the need for "long-distance constraints" such as unification variables. There has been considerable interest in algorithms of this kind by researchers from the functional programming community recently, notably by Pierce and Turner [100], who presented a local type inference algorithm for a type system with subtyping and parametric polymorphism, which was too difficult to handle for conventional type inference techniques. Recent work by Pfenning [96] shows that the idea of bidirectional type checking can also be applied successfully in the field of logical frameworks.

2.4.3 A static compression algorithm

We will now examine which of the restrictions made in the proof of theorem 2.2 can be relaxed, while still being able to solve the constraints arising during reconstruction using pattern unification. An important question is which parts of the argument used in the proof of theorem 2.2 still work if we have only pattern terms instead of ground terms, for example if the proposition ψ against which the inferred proposition φ is to be checked is a pattern instead of a ground term. In this case, it is no longer guaranteed that when solving the constraints for $(h_i \cdot t)$, where $(h_i : \varphi) \in \Gamma$, we can compute a substitution θ, such that $\theta(\varphi)$ is ground. If $\theta(\varphi)$ is a pattern and $\langle \{\varphi =^? \bigwedge x.\ ?f\ \overline{V_\Gamma}\ x\} \mid \theta \rangle \longrightarrow^*_{\mathcal{S}} (\{\},\ \theta')$, then $\theta'(?f\ \overline{V_\Gamma})$ is a pattern, but the proposition $\theta'(?f\ \overline{V_\Gamma}\ t)$ synthesized for the proof $(h_i \cdot t)$ will not necessarily be a pattern.

In particular, we want to relax condition 2 and replace at least some of the term arguments t_i in proofs of the form $(c \cdot t_1 \cdot \cdots \cdot t_n)$ by placeholders. Clearly, if $t_i = \_$ for some i, we are no longer able to infer a ground term for $(c \cdot t_1 \cdot \cdots \cdot t_n)$. To ensure that we can at least infer a pattern term and therefore do not generate constraints which are unsolvable using pattern unification, we may only omit specific arguments of c. The approach pursued in this section is to find out *statically* using information from the signature Σ, i.e. by examining the shape of the proposition $\Sigma(c)$, which arguments can safely be omitted. In general, $\Sigma(c)$ has the form $(\bigwedge a_1 \ldots a_n.\ \varphi)$, where φ, which we call the *body* of $\Sigma(c)$, does not start with a $\bigwedge$ quantifier and $a_1 \ldots a_n$ are called the *parameters* of c. A concept which is crucial for the solvability of proof reconstruction is that of a *strict occurrence* of a parameter. The notion of *strictness* has been introduced by Pfenning and Schürmann [98] and closely corresponds to the notion of a *pattern* introduced in §2.4.1. It is also related to the *rigid path* criterion introduced by Huet in the context of full higher order unification [52].

Definition 2.4 (Strict occurrence) Let $\mathcal{A} = \{a_1, \ldots, a_n\}$. Then the sets of parameters with strict and non-strict occurrences are defined by the two functions

$$\begin{array}{lcll}
\mathsf{strict}_{\mathcal{A}}\ (a\ x_1\ \cdots\ x_n) & = & \{a\} & \text{where } a \in \mathcal{A},\ x_i \in \mathcal{V}\backslash\mathcal{A} \text{ and } x_i \neq x_j \text{ for } i \neq j \\
\mathsf{strict}_{\mathcal{A}}\ (a\ t_1\ \cdots\ t_n) & = & \{\} & \text{where } a \in \mathcal{A} \\
\mathsf{strict}_{\mathcal{A}}\ (b\ t_1\ \cdots\ t_n) & = & \bigcup_{1 \leq i \leq n} \mathsf{strict}_{\mathcal{A}}\ (t_i) & \text{where } b \notin \mathcal{A} \\
\mathsf{strict}_{\mathcal{A}}\ (\lambda x.\ t) & = & \mathsf{strict}_{\mathcal{A}}\ (t) &
\end{array}$$

$$\begin{array}{lcll}
\mathsf{nstrict}_{\mathcal{A}}\ (a\ x_1\ \cdots\ x_n) & = & \{\} & \text{where } a \in \mathcal{A},\ x_i \in \mathcal{V}\backslash\mathcal{A} \text{ and } x_i \neq x_j \text{ for } i \neq j \\
\mathsf{nstrict}_{\mathcal{A}}\ (a\ t_1\ \cdots\ t_n) & = & \mathit{Vars}(a\ t_1\ \cdots\ t_n) \cap \mathcal{A} & \text{where } a \in \mathcal{A} \\
\mathsf{nstrict}_{\mathcal{A}}\ (b\ t_1\ \cdots\ t_n) & = & \bigcup_{1 \leq i \leq n} \mathsf{nstrict}_{\mathcal{A}}\ (t_i) & \text{where } b \notin \mathcal{A} \\
\mathsf{nstrict}_{\mathcal{A}}\ (\lambda x.\ t) & = & \mathsf{nstrict}_{\mathcal{A}}\ (t) &
\end{array}$$

We say that a parameter $a \in \mathcal{A}$ has a *strict occurrence* in t, where t is in *β-normal form*, if $a \in \mathsf{strict}_{\mathcal{A}}\ t$, whereas it has a *non-strict occurrence* if $a \in \mathsf{nstrict}_{\mathcal{A}}\ t$.

In other words, an occurrence of a parameter a is called strict, if

1. it has only distinct bound variables as arguments, and
2. it does not occur in the argument of another parameter

For example, in the rule

$$\mathit{exE} : \bigwedge P\ Q.\ \mathsf{Tr}\ (\exists x.\ P\ x) \Longrightarrow (\bigwedge x.\ \mathsf{Tr}\ (P\ x) \Longrightarrow \mathsf{Tr}\ Q) \Longrightarrow \mathsf{Tr}\ Q$$

all occurrences of P and Q are strict, whereas in the rule

$$\mathit{exI} : \bigwedge P\ x.\ \mathsf{Tr}\ (\overbrace{P\ x}^{\text{non-strict}}) \Longrightarrow \mathsf{Tr}\ (\exists x.\ \overbrace{P}^{\text{strict}}\ x)$$

only the occurrence of P in the conclusion is strict, since its argument x is locally bound by an existential quantifier. In the premise, both the occurrences of P and x are non-strict: the occurrence of P is non-strict, because it has the parameter x as an argument, and the occurrence of x is non-strict, too, since it is in the argument of the parameter P. Inference rules of propositional logic, such as

$$\mathit{disjE} : \quad \bigwedge P\ Q\ R.\ \mathsf{Tr}\ (P \vee Q) \Longrightarrow (\mathsf{Tr}\ P \Longrightarrow \mathsf{Tr}\ R) \Longrightarrow (\mathsf{Tr}\ Q \Longrightarrow \mathsf{Tr}\ R) \Longrightarrow \mathsf{Tr}\ R$$

contain only strict occurrences of parameters.

Due to the close correspondence between the concept of a pattern and that of a strict occurrence, it easily follows that by replacing all parameters in φ having only strict occurrences by unification variables, and any other parameters by ground terms, a pattern is obtained. Therefore, if $\Sigma(c) = (\bigwedge a_1 \ldots a_n.\ \varphi)$ and for all i with $t_i = \_$, we have that $a_i \notin \mathsf{nstrict}_{\mathcal{A}}\ \varphi$, then we can synthesize a proposition for $(c \cdot t_1 \cdots t_n)$ which is a pattern, i.e. if

$$\Gamma \vdash c \cdot t_1 \cdots t_n \rhd (c \cdot t'_1 \cdots t'_n,\ \varphi,\ C) \text{ and } (C,\ \{\}) \longrightarrow^*_{\mathcal{S}} (\{\},\ \theta)$$

then $\theta(\varphi)$ is a pattern.

It is not immediately obvious why condition 2 in the above definition of strictness is needed. To see why it is necessary, note that a parameter, say x, which occurs in the argument of another parameter, say P, as in the term $P\ x$, may not in general be instantiated with a unification variable if the resulting term is supposed to be a pattern, even if the parameter x has only distinct bound variables as arguments. As an example, consider again the rule *exI* above. Now assume that P is replaced by the ground term $(\lambda x :: \tau \Rightarrow \sigma.\ d\ (x\ c))$, where c and d are constants with $\Sigma(c) = \tau$ and $\Sigma(d) = \sigma \Rightarrow \mathsf{bool}$. If x is now replaced by a unification variable $?x_{\tau\Rightarrow\sigma}$, the rule *exI* is turned into

$$\mathsf{Tr}\ (d\ (?x\ c)) \Longrightarrow \mathsf{Tr}\ (\exists x.\ d\ (x\ c))$$

which is a non-pattern term, since $?x$ has been projected into a head position. Note that such a situation can only arise if the variable x in $P\ x$ has a *function type*. This means that e.g. the parameter $n :: \mathit{nat}$ occurring in the conclusion $P\ (n :: \mathit{nat})$ of the induction rule for natural numbers (see also §4.3.4) could be omitted from the proof, although the occurrence of n is not strict.

Unfortunately, the strictness criterion alone is not enough to determine the parameters of an inference rule which can safely be omitted, since already the omission of parameters which have only strict occurrences can give rise to constraints which are not solvable using pattern unification. This may happen for inference rules r, where the body of $\Sigma(r)$ contains a meta level universal quantifier "$\bigwedge$". As an example, consider an inference rule r of the form

$$r : \bigwedge P.\ \ ((\bigwedge x.\ \mathsf{Tr}\ (P\ x)) \Longrightarrow \cdots) \Longrightarrow \cdots$$

where P has only a strict occurrence. Now consider the proof

$$r \cdot \_ \bullet (\boldsymbol{\lambda} h : \_.\ \ \cdots (h \cdot t) \cdots)$$

If we cannot infer a ground term for the omitted parameter P of rule r from the context, but only a pattern term, we can only infer a pattern term for the proposition associated with h, too. Hence, for reasons which have already been explained at the beginning of this section, the subproof $(h \cdot t)$ may give rise to a non-pattern constraint. Similar problems can arise in connection with rules containing *predicate variables* of type $\overline{\alpha} \Rightarrow \mathsf{prop}$. For example, consider the rule

$$\mathit{thin} : \bigwedge(P :: \mathsf{prop})\ (Q :: \mathsf{prop}).\ \ P_{\mathsf{prop}} \Longrightarrow Q_{\mathsf{prop}} \Longrightarrow Q_{\mathsf{prop}}$$

which might be used in a proof of the form

$$\mathit{thin} \cdot \_ \cdot \_ \bullet (\boldsymbol{\lambda} h : \_.\ \ \cdots (h \cdot t) \cdots)$$

Due to the structure of the rule *thin*, it will not in general be possible to infer a ground term for the omitted argument P, if just the proposition corresponding to the above proof is given. As in the previous example, the subproof $(h \cdot t)$ may therefore give rise to a non-pattern constraint. Note that the above problem can only be caused by predicate variables which appear on the "top level" of a formula, i.e. as immediate subformulae of $\varphi \Longrightarrow \psi$ or $(\bigwedge x.\ \varphi)$. Predicate variables appearing as arguments of other constants, such as the variables P_{prop} and Q_{prop} in the meta equality $P_{\mathsf{prop}} \equiv Q_{\mathsf{prop}}$ are unproblematic, since another rule, such as $\equiv$-elimination would have to be applied first, before any $\bigwedge$ quantifiers occurring in P or Q could be eliminated, which could then produce a non-pattern constraint. This idea is captured by the following definition:

Definition 2.5 (Top level predicate variables) The set $\mathsf{propv}_{\mathcal{A}}\ (\varphi)$ of *top level predicate variables* of a proposition φ is defined as follows:

$$\begin{array}{lcll}
\mathsf{propv}_{\mathcal{A}}\ (\varphi \Longrightarrow \psi) & = & \mathsf{propv}_{\mathcal{A}}(\varphi) \cup \mathsf{propv}_{\mathcal{A}}(\psi) & \\
\mathsf{propv}_{\mathcal{A}}\ (\bigwedge x.\ \varphi) & = & \mathsf{propv}_{\mathcal{A}}(\varphi) & \\
\mathsf{propv}_{\mathcal{A}}\ (P_{\overline{\alpha} \Rightarrow \mathsf{prop}}\ \overline{t}) & = & \{P_{\overline{\alpha} \Rightarrow \mathsf{prop}}\} & \text{if } P \in \mathcal{A} \\
\mathsf{propv}_{\mathcal{A}}\ (\varphi) & = & \{\} & \text{otherwise}
\end{array}$$

We will now show that the problematic situations described above can only be caused by inference rules r, where the body of $\Sigma(r)$ contains either a $\bigwedge$ quantifier in a *positive* position, or top level predicate variables.

Theorem 2.3 If p is a proof containing a subproof of the form $(h \cdot \cdots \cdot t)$, i.e. an application of a hypothesis to a term, and $\Gamma \vdash p : \varphi$, then one of the following conditions must be satisfied:

1. there is some $(h_i : \psi_i) \in \Gamma$ such that $\bigwedge$ occurs *positively* in ψ_i
2. $\bigwedge$ occurs *negatively* in φ
3. p contains some proof constant r such that the body of $\Sigma(r)$ either contains a *positive* occurrence of $\bigwedge$, or contains *top level predicate variables.*

Proof: by induction on the size of p, which we assume to be in normal form. We need to consider the following cases:

- p is an application.
  - If the head of p is a hypothesis variable, then p must either have the form $(h_i \cdot \overline{p} \cdot t \cdots)$ or $(h_i \cdot p_1 \cdot \cdots \cdot p_n)$ where some p_i contains an application of a hypothesis to a term. In the former case, we have $\varphi = (\cdots \Longrightarrow (\bigwedge x.\ \varphi'\ x))$, i.e. $\bigwedge$ occurs positively in φ. In the latter case, note that $\Gamma \vdash p_i : \varphi_i$. By induction hypothesis, either $\bigwedge$ occurs positively in some ψ_j with $(h_j : \psi_j) \in \Gamma$, or p_i contains a proof constant r of the kind described in condition 3 above, or $\bigwedge$ occurs negatively in φ_i. In the first two cases, the claim follows directly. In the last case, note that we must have some $(h_i : \psi_i) \in \Gamma$ where

    $$\psi_i = (\varphi_1 \Longrightarrow \cdots \Longrightarrow \varphi_i \Longrightarrow \cdots \Longrightarrow \varphi_n \Longrightarrow \psi)$$

    Hence, since $\bigwedge$ occurs negatively in φ_i, it must occur positively in ψ_i.
  - If the head of p is a constant r, then $p = (r \cdot \overline{t} \cdot \cdots \cdot q \cdots)$, where q contains an application of a hypothesis to a term. Now the argument is similar to the one used in the previous case: if, by induction hypothesis, we have $\Gamma \vdash q : \varphi'$ and $\bigwedge$ occurs negatively in φ', then $\Sigma(r)$ must have one of the following two forms:
    * $\Sigma(r) = (\bigwedge \overline{a}.\ \cdots \Longrightarrow \cdots \Longrightarrow a_i\ \overline{x})$ and $t_i = (\lambda \overline{x}.\ \varphi' \Longrightarrow \cdots)$, or
    * $\Sigma(r) = (\bigwedge \overline{a}.\ \cdots \Longrightarrow \varphi'' \Longrightarrow \cdots)$, where $\varphi''\{\overline{a} \mapsto \overline{t}\} =_\beta \varphi'$

    In the latter case, if φ'' contains no top level predicate variables, the $\bigwedge$ quantifiers in φ'' must occur in exactly the same positions as they do in φ'. Since $\bigwedge$ occurs negatively in φ', it must therefore occur positively in the body of $\Sigma(r)$
- p is an abstraction, i.e.

$$p = (\boldsymbol{\lambda}(\overline{x_1} :: \overline{\tau_1})\ (h'_1 : \psi'_1)\ \ldots\ (\overline{x_n} :: \overline{\tau_n})\ (h'_n : \psi'_n)\ (\overline{x_{n+1}} :: \overline{\tau_{n+1}}).\ q)$$

For p to be correct, we must have $\Gamma, \Gamma' \vdash q : \varphi'$, where

$$\Gamma' = \overline{x_1} :: \overline{\tau_1},\ h'_1 : \psi'_1,\ \ \ldots,\ \overline{x_n} :: \overline{\tau_n},\ h'_n : \psi'_n,\ \overline{x_{n+1}} :: \overline{\tau_{n+1}}$$

and

$$\varphi = (\bigwedge \overline{x_1} :: \overline{\tau_1}.\ \psi'_1 \implies (\cdots \implies (\bigwedge \overline{x_n} :: \overline{\tau_n}.\ \psi'_n \implies (\bigwedge \overline{x_{n+1}} :: \overline{\tau_{n+1}}.\ \varphi'))\cdots))$$

By induction hypothesis, we either have that $\bigwedge$ occurs negatively in φ', or q contains a proof constant r of the kind described in condition 3 above, or $(h : \psi) \in \Gamma \cup \Gamma'$, where $\bigwedge$ occurs positively in ψ. In the first case, it easily follows that $\bigwedge$ must also occur negatively in φ, whereas in the second case, the claim follows immediately. In the last case, we have to distinguish whether $(h : \psi) \in \Gamma$ or $(h : \psi) \in \Gamma'$. In the former case, the claim follows immediately. In the latter case, $h = h'_i$ and $\psi = \psi'_i$ for some i, where $\bigwedge$ occurs positively in ψ'_i. Hence, $\bigwedge$ must occur negatively in φ.

Based on the above considerations, we can now specify a criterion to statically determine the parameters of an inference rule, which can safely be omitted without producing any non-pattern constraints during reconstruction. To this end, we introduce the concept of an *unsafe occurrence* of a parameter, which is a strengthened version of the concept of *non-strictness* presented above.

Definition 2.6 (Unsafe occurrence) Let $\mathcal{A} = \{a_1, \ldots, a_n\}$ be a set of parameters. Then the set of parameters with *unsafe occurrences* in a formula is defined by

$$\begin{array}{lcll}
\mathsf{unsafe}^{+}_{\mathcal{A}}\ \mathcal{Q}\ (\bigwedge y.\ \varphi) & = & \mathsf{unsafe}^{+}_{\mathcal{A}}\ (\{y\} \cup \mathcal{Q})\ (\varphi) & \\
\mathsf{unsafe}^{-}_{\mathcal{A}}\ \mathcal{Q}\ (\bigwedge y.\ \varphi) & = & \mathsf{unsafe}^{-}_{\mathcal{A}}\ \mathcal{Q}\ (\varphi) & \\
\mathsf{unsafe}^{\pi}_{\mathcal{A}}\ \mathcal{Q}\ (\varphi \Longrightarrow \psi) & = & \mathsf{unsafe}^{-\pi}_{\mathcal{A}}\ \mathcal{Q}\ (\varphi) \cup \mathsf{unsafe}^{\pi}_{\mathcal{A}}\ \mathcal{Q}\ (\psi) & \\
\mathsf{unsafe}^{\pi}_{\mathcal{A}}\ \mathcal{Q}\ (P_{\overline{\alpha} \Rightarrow \mathsf{prop}}\ t_1\ \cdots\ t_n) & = & \mathit{Vars}(P_{\overline{\alpha} \Rightarrow \mathsf{prop}}\ t_1\ \cdots\ t_n) \cap \mathcal{A} & \text{if } P \in \mathcal{A} \cup \mathcal{Q} \\
\mathsf{unsafe}^{\pi}_{\mathcal{A}}\ \mathcal{Q}\ (\varphi) & = & \mathsf{nstrict}_{\mathcal{A} \cup \mathcal{Q}}\ (\varphi) & \text{otherwise}
\end{array}$$

where $\pi \in \{-, +\}$ is called the *polarity* of the formula, and

$$-\pi \ = \ \begin{cases} - & \text{if } \pi = + \\ + & \text{if } \pi = - \end{cases}$$

We say that a parameter $a \in \mathcal{A}$ has an *unsafe occurrence* in a formula φ, where φ is in *β-normal form*, if $a \in \mathsf{unsafe}^{+}_{\mathcal{A}}\ \{\}\ (\varphi)$.

The concept of a *safe occurrence* could be defined analogously. In other words, an occurrence of a parameter is unsafe, if

1. it is non-strict, or
2. it has a variable bound by a positive $\bigwedge$ quantifier as an argument, or
3. it occurs in the argument of a variable bound by a positive $\bigwedge$ quantifier, or
4. it is a top level predicate variable

Due to theorem 2.3, the additional restrictions concerning predicate variables and variables bound by $\bigwedge$ quantifiers guarantee that we can always reconstruct the proposition φ corresponding to a hypothesis variable $(h : \varphi) \in \Gamma$, which could be subject to $\bigwedge$-elimination. As a consequence, given an inference rule r with $\Sigma(r) = (\bigwedge a_1 \ldots a_n.\ \varphi)$ and a proof term $(r \cdot t_1 \cdot \cdots \cdot t_n)$, we may safely omit any argument t_i for which $a_i \notin \mathsf{unsafe}^{+}_{\mathcal{A}}\ \{\}\ (\varphi)$, where $\mathcal{A} = \{a_1, \ldots, a_n\}$.

At first sight, the additional conditions required for a parameter in order to be *safe* might seem overly restrictive. However, it turns out in practice that most inference rules of standard object logics do not involve deeper nestings of the meta level connectives $\bigwedge$ and $\Longrightarrow$, and therefore do not contain positive occurrences of $\bigwedge$, apart from the outermost quantifiers binding the parameters. For example, the quantifier $(\bigwedge x.\ \ldots)$ occurs *negatively* in the rule *exE* given above. An example for a rule with a quantifier having a *positive* occurrence is the induction rule for the *accessible part* of a relation (see also §4.3.3)

$$\begin{array}{l} \mathit{acc.induct}:\ \bigwedge z\ r\ P.\ \mathsf{Tr}\ (x \in \mathit{acc}\ r) \Longrightarrow (\bigwedge x.\ (\bigwedge y.\ \mathsf{Tr}\ ((y,\ x) \in r) \Longrightarrow \mathsf{Tr}\ (y \in \mathit{acc}\ r)) \Longrightarrow \\ \qquad (\bigwedge y.\ \mathsf{Tr}\ ((y,\ x) \in r) \Longrightarrow \mathsf{Tr}\ (P\ y)) \Longrightarrow \mathsf{Tr}\ (P\ x)) \Longrightarrow \mathsf{Tr}\ (P\ z) \end{array}$$

where the quantifier $\bigwedge y$ occurs positively. Since in the subterm $P\ y$, the parameter P has the quantified variable y as an argument, it may not be omitted. Note that we would not even have been allowed to omit P if $\bigwedge y$ had had no positive occurrence, since in the subterm $P\ z$, the parameter P has another parameter z as an argument. Rules like the above are hard to work with in connection with old-style tactic scripts, because the standard resolution tactic only allows the elimination of object-level connectives such as $\longrightarrow$ and $\forall$, whereas there is no direct way of eliminating the meta-level connectives $\Longrightarrow$ and $\bigwedge$ occurring in the premises introduced by *acc.induct*. Therefore, such rules will seldom be encountered in older Isabelle theories, but may occur in newer theories based on Isar, which allows for a more smooth handling of formulae containing meta-level connectives [120, §5.2.5]. Rules involving top level predicate variables such as

$$\mathit{cut} : \bigwedge (P :: \mathsf{prop})\ (Q :: \mathsf{prop}).\ (P_{\mathsf{prop}} \Longrightarrow Q_{\mathsf{prop}}) \Longrightarrow P_{\mathsf{prop}} \Longrightarrow Q_{\mathsf{prop}}$$

or the *thin* rule given above can easily be eliminated by expanding their derivations. While the expansion of *cut* may occasionally lead to a slightly larger proof, expanding the derivation

$$\boldsymbol{\lambda}(P :: \mathsf{prop})\ (Q :: \mathsf{prop})\ (h_1 : P_{\mathsf{prop}})\ (h_2 : Q_{\mathsf{prop}}).\ h_2$$

of *thin* is clearly advantageous, since this, in connection with proof normalization, removes the superfluous proof of P_{prop} from the proof term. Top level predicate variables also occur in the introduction and elimination rules for meta-equality on propositions (see also §3)

$$\begin{array}{l} \mathit{eqI} : \bigwedge (P :: \mathsf{prop})\ (Q :: \mathsf{prop}).\ (P_{\mathsf{prop}} \Longrightarrow Q_{\mathsf{prop}}) \Longrightarrow (Q_{\mathsf{prop}} \Longrightarrow P_{\mathsf{prop}}) \Longrightarrow P_{\mathsf{prop}} \equiv Q_{\mathsf{prop}} \\ \mathit{eqE} : \bigwedge (P :: \mathsf{prop})\ (Q :: \mathsf{prop}).\ P_{\mathsf{prop}} \equiv Q_{\mathsf{prop}} \Longrightarrow P_{\mathsf{prop}} \Longrightarrow Q_{\mathsf{prop}} \end{array}$$

In specific situations, which are described in more detail in §3.3.1, these rules can (and should) be eliminated from a proof term as well. Inference rules for object logics, i.e. the majority of the rules used in the construction of proofs, usually do not contain any meta level predicate variables of type $\overline{\tau} \Rightarrow \mathsf{prop}$ at all, but only object level predicate variables of type $\overline{\tau} \Rightarrow \mathsf{bool}$, where bool is the type of object level truth values. Such variables may only occur in the argument of a coercion function $\mathsf{Tr} :: \mathsf{bool} \Rightarrow \mathsf{prop}$, as in $\mathsf{Tr}\ P_{\mathsf{bool}}$.

Based on the above definition of unsafe, we can now develop a static compression algorithm. The algorithm is given by a compression judgement $\Gamma \vdash p \gg (p_p,\ r)$, which is characterized

$$\frac{\Sigma(c) = \bigwedge \overline{a}.\ \varphi}{\Gamma \vdash c_{\{\overline{\alpha} \mapsto \overline{\tau}\}} \gg (c_{\{\overline{\alpha} \mapsto \_\}},\ \mathsf{recipe}_{\overline{a},\varphi}\ \overline{a})} \qquad \frac{}{\Gamma \vdash h \gg (h,\ [])}$$

$$\frac{\Gamma \vdash p \gg (p_p,\ -.r)}{\Gamma \vdash p \cdot t \gg (p_p \cdot \_,\ r)} \qquad \frac{\Gamma \vdash p \gg (p_p,\ \bullet.r)}{\Gamma \vdash p \cdot t \gg (p_p \cdot t,\ r)} \qquad \frac{\Gamma \vdash p \gg (p_p,\ [])}{\Gamma \vdash p \cdot t \gg (p_p \cdot t,\ [])}$$

$$\frac{\Gamma \vdash p \gg (p_p,\ r) \quad \Gamma \vdash q \gg (q_p,\ r')}{\Gamma \vdash p \boldsymbol{\cdot} q \gg (p_p \boldsymbol{\cdot} q_p,\ [])}$$

$$\frac{\Gamma, x :: \tau \vdash p \gg (p_p,\ r)}{\Gamma \vdash (\boldsymbol{\lambda} x :: \tau.\ p) \gg (\boldsymbol{\lambda} x :: \_.\ p_p,\ [])} \qquad \frac{\Gamma, h : \varphi \vdash p \gg (p_p,\ r)}{\Gamma \vdash (\boldsymbol{\lambda} h : \varphi.\ p) \gg (\boldsymbol{\lambda} h : \_.\ p_p,\ [])}$$

$$r \;=\; -.r \mid \bullet.r \mid []$$

$$\begin{array}{lcl} \mathsf{recipe}_{\mathcal{A},\varphi}\ [] & = & [] \\ \mathsf{recipe}_{\mathcal{A},\varphi}\ (a, \overline{a}) & = & \begin{cases} \bullet.\mathsf{recipe}_{\mathcal{A},\varphi}\ \overline{a} & \text{if } a \in \mathsf{unsafe}^{+}_{\mathcal{A}}\ \{\}\ (\varphi) \\ -.\mathsf{recipe}_{\mathcal{A},\varphi}\ \overline{a} & \text{otherwise} \end{cases} \end{array}$$

Figure 2.6: Static compression algorithm

by the inference rules shown in Figure 2.6. Given a context Γ and a proof p, the compression judgement returns a compressed proof p_p. To indicate which term arguments of p_p may be omitted, the judgement also returns a so-called *representation recipe*[5] r. A representation recipe is essentially a list of elements from the set $\{-, \bullet\}$, where "$-$" means that the corresponding argument may be omitted, while "$\bullet$" means that the argument has to be retained. The representation recipe is only relevant when compressing proofs of the form $(p \cdot t)$, i.e. applications of proofs to terms. Representation recipes are generated by the rule for compressing *proof constants* c, which produces a recipe having the same length as the parameter list $\overline{a}$ of c, where $\Sigma(c) = \bigwedge \overline{a}.\ \varphi$. For example, the representation recipes for the rules *exE*, *exI*, and *disjE* are $-.-$, $\bullet.\bullet$, and $-.-.-$, respectively. Representation recipes are decomposed when compressing the term argument list of a proof constant. When compressing the proof $(((c \cdot \overline{t}) \cdot t) \cdot \overline{t'})$, the representation recipe $\varrho.r$ returned by the reconstruction judgement for the subproof $(c \cdot \overline{t})$ is first decomposed into its head element $\varrho \in \{-, \bullet\}$, which determines whether or not the argument t is to be omitted, and a residual representation recipe r, which is then used for the compression of the subsequent arguments $\overline{t'}$. Note that the compression rule for proof variables h just returns the dummy representation recipe $[]$, which means that none of the arguments of h may be omitted.

2.4.4 A refined strategy for omitting terms

So far, the strategy used for omitting terms in proofs has been an "all or nothing" approach, i.e. either a term may be completely omitted, or it may not be omitted at all. To improve this situation, we now refine the grammars and the reconstruction judgements for partial proofs p_p and terms t_p presented in Figure 2.5, such that placeholders "$\_$" standing for terms may not just occur as immediate arguments of the proof term constructors for $\Longrightarrow$-introduction and $\bigwedge$-elimination, i.e. in $(\boldsymbol{\lambda} h : \_.\ p_p)$ and $(p_p \cdot \_)$, but may occur anywhere in a term. For example,

[5]The term *representation recipe* was coined by Necula and Lee [76, 77].

we would like to be able to construct a partial proof term of the form

$$\mathit{subst} \cdot (\lambda x.\ c\ x\ \_) \cdot s \cdot t \bullet p_1 \bullet p_2$$

where

$$\mathit{subst} : \bigwedge P\ s\ t.\ s = t \Longrightarrow P\ s \Longrightarrow P\ t$$

and c is a constant of type $\alpha \Rightarrow \beta \Rightarrow \mathit{bool}$. This allows for a more fine-grained proof compression strategy, which just retains those parts of a term which are sufficient to guarantee that only pattern constraints arise during reconstruction, instead of keeping the whole (possibly large) term in the proof object.

As has been noted in §2.4.1, placeholders may depend on variables bound in the context, which is why unification variables inserted for placeholders during reconstruction have to be *lifted*. However, when looking at the above example involving the *subst* rule, we easily notice that lifting a unification variable, such that it depends on the bound variable x causes a problem. If we turn the partial term $(\lambda x.\ c\ x\ \_)$ into the placeholder-free term $(\lambda x.\ c\ x\ (?v\ x))$, then an attempt to reconstruct the above proof will inevitably lead to the non-pattern constraint $c\ s\ (?v\ s) =^? \varphi$, where φ is the proposition synthesized for the subproof p_2. We therefore stipulate that placeholders occurring in a term may not depend on any variables bound by *outer* abstractions, i.e. in a partial term $t_p = (\lambda \overline{x}.\ u_p)$, none of the placeholders occurring in u_p may depend on any of the variables in $\overline{x}$. The same restrictions apply to variables which are bound by outer abstractions of a subterm, which is an argument of a variable bound by an outer abstraction, e.g. in a partial term of the form $t_p = (\lambda \overline{x}. \ldots\ (x_i\ (\lambda \overline{y}.\ u_p))\ \ldots)$, where $x_i \in \overline{x}$, none of the placeholders occurring in u_p may depend on variables in $\overline{x} \cup \overline{y}$. To understand the intuition behind this restriction, consider e.g. the terms $f = (\lambda x.\ x\ (\lambda y.\ ?v\ y))$ and $g = (\lambda z.\ z\ c)$, which are patterns, but the application $\downarrow_\beta (f\ g) = ?v\ c$ is *not*. Consequently, the compression algorithm may not replace any subterm of a term to be compressed by a placeholder, which contains a variable bound by an outer abstraction. In contrast, a placeholder occurring in the subterm u_p of the partial term $t_p = c\ (\lambda \overline{x}.\ u_p)$ may depend on any of the variables in $\overline{x}$. Note that this compression strategy crucially relies on the fact that constants have to be unfolded explicitely. In the presence of implicit "on-the-fly" expansion of constants, a detailed analysis of the definition of c occurring in the above term t_p would be required, in order to decide whether one may introduce placeholders in u_p depending on the bound variable x. Besides, such a constant expansion mechanism may also lead to other subtle problems in connection with unification, as e.g. described by Pfenning and Schürmann [98].

We will now formalize what we have just explained informally. First of all, we define a class of terms, which, when substituted for variables in a term t, turn t into a pattern term. Essentially, we define a subset of higher-order patterns, which are *closed under application*. Note that although the term $P = (\lambda x.\ c\, x\ (?v\ x))$ from the above example is a pattern term, its application $(P\ s)$ to another term s is usually *not* a pattern[6], since after β-reduction, the unification variable $?v$ will have the term s as an argument. The purpose of the two mutually recursive judgements $\vdash_{outer}$ and $\vdash_{inner}$ presented in Figure 2.7 is now exactly to rule out terms like P above. Each of the judgements involves two variable contexts Γ_1 and Γ_2 containing variables bound by "*outer*" and "*inner*" abstractions, respectively. These judgements enforce that none of the unification variables occurring in a term t, for which $\Gamma_1 \mid \Gamma_2 \vdash_{outer} t$ or $\Gamma_1 \mid \Gamma_2 \vdash_{inner} t$, may have variables from Γ_1 as arguments. Moreover, if $\Gamma_1 \mid \Gamma_2 \vdash_{outer} (\lambda x :: \tau.\ u)$, none of

[6] assuming that s is not a bound variable

$$\frac{\Gamma_1, x :: \tau \mid \Gamma_2 \vdash_{outer} t}{\Gamma_1 \mid \Gamma_2 \vdash_{outer} \lambda x :: \tau.\ t} \qquad \frac{\Gamma_1 \mid \Gamma_2 \Vdash_{inner} \overline{t} \quad z \notin \overline{V_{\Gamma_1}} \quad z \notin \mathcal{UV}}{\Gamma_1 \mid \Gamma_2 \vdash_{outer} z\ \overline{t}} \qquad \frac{\Gamma_1 \mid \Gamma_2 \Vdash_{outer} \overline{t} \quad z \in \overline{V_{\Gamma_1}}}{\Gamma_1 \mid \Gamma_2 \vdash_{outer} z\ \overline{t}}$$

$$\frac{\forall i.\ i \neq j \to x_i \neq x_j \quad \forall i.\ x_i \in \mathcal{V} \backslash \overline{V_{\Gamma_1}} \quad \Gamma_1, \Gamma_2 \nvdash ?v\ x_1\ \cdots\ x_n :: \tau \Rightarrow \sigma}{\Gamma_1 \mid \Gamma_2 \vdash_{outer} ?v\ x_1\ \cdots\ x_n}$$

$$\frac{}{\Gamma_1 \mid \Gamma_2 \Vdash_{inner} []} \qquad \frac{\Gamma_1 \mid \Gamma_2 \vdash_{inner} t \quad \Gamma_1 \mid \Gamma_2 \Vdash_{inner} \overline{t}}{\Gamma_1 \mid \Gamma_2 \Vdash_{inner} t, \overline{t}}$$

$$\frac{\Gamma_1 \mid \Gamma_2, x :: \tau \vdash_{inner} t}{\Gamma_1 \mid \Gamma_2 \vdash_{inner} \lambda x :: \tau.\ t} \qquad \frac{\Gamma_1 \mid \Gamma_2 \Vdash_{inner} \overline{t} \quad z \notin \overline{V_{\Gamma_1}} \quad z \notin \mathcal{UV}}{\Gamma_1 \mid \Gamma_2 \vdash_{inner} z\ \overline{t}} \qquad \frac{\Gamma_1 \mid \Gamma_2 \Vdash_{outer} \overline{t} \quad z \in \overline{V_{\Gamma_1}}}{\Gamma_1 \mid \Gamma_2 \vdash_{inner} z\ \overline{t}}$$

$$\frac{\forall i.\ i \neq j \to x_i \neq x_j \quad \forall i.\ x_i \in \mathcal{V} \backslash \overline{V_{\Gamma_1}}}{\Gamma_1 \mid \Gamma_2 \vdash_{inner} ?v\ x_1\ \cdots\ x_n}$$

$$\frac{}{\Gamma_1 \mid \Gamma_2 \Vdash_{outer} []} \qquad \frac{\Gamma_1 \mid \Gamma_2 \vdash_{outer} t \quad \Gamma_1 \mid \Gamma_2 \Vdash_{outer} \overline{t}}{\Gamma_1 \mid \Gamma_2 \Vdash_{outer} t, \overline{t}}$$

Figure 2.7: Higher-order patterns closed under application

the unification variables in u may depend on the variable x bound by the outer abstraction. For example, for the term P given above, we have $\Gamma_1 \mid \Gamma_2 \nvdash_{outer} P$. In contrast, note that unification variables occurring in u', for which $\Gamma_1 \mid \Gamma_2 \vdash_{inner} (\lambda x :: \tau.\ u')$, may have x as an argument. The judgement $\vdash_{outer}$ also rules out terms of the form $(?v\ x_1\ \cdots\ x_n)$, which have a function type, i.e. where $?v :: \tau_1 \Rightarrow \cdots \Rightarrow \tau_{n+1} \Rightarrow \sigma$, since applying them to another term will usually not yield a pattern either. The judgements $\Vdash_{outer}$ and $\Vdash_{inner}$ are just extensions of the judgements $\vdash_{outer}$ and $\vdash_{inner}$ to lists of terms. Moreover, the set of terms characterized by the above judgements is also a subset of the set of β-normal terms. We are now ready to formulate the main properties of $\vdash_{outer}$ and $\vdash_{inner}$:

Theorem 2.4 (Closure under application and substitution)

1. If $\Gamma_1, \Gamma_2 \vdash t : \sigma \Rightarrow \tau$, and $\Gamma_1 \mid \Gamma_2 \vdash_{outer} t$, and $\Gamma_1, \Gamma_2 \vdash u : \sigma$, and $\Gamma_1 \mid \Gamma_2 \vdash_{outer} u$, then $\Gamma_1 \mid \Gamma_2 \vdash_{outer} \downarrow_\beta (t\ u)$

2. If $\Gamma_1, x :: \sigma, \Gamma_2 \vdash t : \tau$, and $\Gamma_1, x :: \sigma \mid \Gamma_2 \vdash_{outer} t$, and $\Gamma_1, \Gamma_2 \vdash u :: \sigma$, and $\Gamma_1 \mid \Gamma_2 \vdash_{outer} u$, then $\Gamma_1 \mid \Gamma_2 \vdash_{outer} \downarrow_\beta (t\{x \mapsto u\})$

3. If $\Gamma_1, x :: \sigma, \Gamma_2 \vdash t : \tau$, and $\Gamma_1, x :: \sigma \mid \Gamma_2 \vdash_{inner} t$, and $\Gamma_1, \Gamma_2 \vdash u :: \sigma$, and $\Gamma_1 \mid \Gamma_2 \vdash_{outer} u$, then $\Gamma_1 \mid \Gamma_2 \vdash_{inner} \downarrow_\beta (t\{x \mapsto u\})$

4. If $\Gamma_1 \mid \Gamma_2 \vdash_{outer} t$, then $\Gamma_1 \mid \Gamma_2 \vdash_{inner} t$

5. If $\Gamma_1 \mid \Gamma_2 \vdash_{outer} t$ or $\Gamma_1 \mid \Gamma_2 \vdash_{inner} t$, then $t \in \mathcal{P}at$

The properties 1 to 3 can be proved by an argument similar to the one used in proofs of weak normalization for the simply-typed λ-calculus, as described e.g. by Matthes and Joachimski [56]. One first proves properties 2 and 3 by main induction on the type σ and side induction

$$t_p, u_p, \varphi_p, \psi_p \ = \ x \mid c_{\{\overline{\alpha} \mapsto \overline{\tau}\}} \mid t_p\ u_p \mid \lambda x :: \tau.\ t_p \mid c_{\{\overline{\alpha} \mapsto \_\}} \mid \lambda x :: \_.\ t_p \mid \_$$

$$\frac{(x :: \tau) \in \Gamma_1 \cup \Gamma_2}{\Gamma_1 \mid \Gamma_2 \vdash x \rhd (x,\ \tau,\ \{\})}\ \mathsf{Var} \qquad \frac{\Sigma(c) = \tau}{\Gamma_1 \mid \Gamma_2 \vdash c_{\{\overline{\alpha} \mapsto \overline{\tau}\}} \rhd (c_{\{\overline{\alpha} \mapsto \overline{\tau}\}},\ \tau\{\overline{\alpha} \mapsto \overline{\tau}\},\ \{\})}\ \mathsf{Const}$$

$$\frac{\Gamma_1, x :: \tau \mid \Gamma_2 \vdash t_p \rhd_{outer} (t,\ \sigma,\ C)}{\Gamma_1 \mid \Gamma_2 \vdash (\lambda x :: \tau.\ t_p) \rhd_{outer} ((\lambda x :: \tau.\ t),\ \tau \Rightarrow \sigma,\ C)}\ \mathsf{Abs}_{outer}$$

$$\frac{\Gamma_1 \mid \Gamma_2, x :: \tau \vdash t_p \rhd_{inner} (t,\ \sigma,\ C)}{\Gamma_1 \mid \Gamma_2 \vdash (\lambda x :: \tau.\ t_p) \rhd_{inner} ((\lambda x :: \tau.\ t),\ \tau \Rightarrow \sigma,\ C)}\ \mathsf{Abs}_{inner}$$

$$\frac{\Gamma_1 \mid \Gamma_2 \vdash t_p \rhd (t,\ \varrho,\ C) \quad \Gamma_1 \mid \Gamma_2 \vdash u_p \rhd_{outer} (u,\ \tau,\ D) \quad t_p = x\ \overline{s_p} \quad x \in \overline{V_{\Gamma_1}}}{\Gamma_1 \mid \Gamma_2 \vdash (t_p\ u_p) \rhd ((t\ u),\ ?\sigma,\ \{\varrho =^? \tau \Rightarrow ?\sigma\} \cup C \cup D)}\ \mathsf{App}_{outer}$$

$$\frac{\Gamma_1 \mid \Gamma_2 \vdash t_p \rhd (t,\ \varrho,\ C) \quad \Gamma_1 \mid \Gamma_2 \vdash u_p \rhd_{inner} (u,\ \tau,\ D) \quad t_p = a\ \overline{s_p} \quad a \in \overline{V_{\Gamma_2}} \cup \mathcal{C}}{\Gamma_1 \mid \Gamma_2 \vdash (t_p\ u_p) \rhd ((t\ u),\ ?\sigma,\ \{\varrho =^? \tau \Rightarrow ?\sigma\} \cup C \cup D)}\ \mathsf{App}_{inner}$$

$$\frac{}{\Gamma_1 \mid \Gamma_2 \vdash \_ \rhd (?v_{\overline{\tau_{\Gamma_2}} \Rightarrow ?\alpha}\ \overline{V_{\Gamma_2}},\ ?\alpha,\ \{\})}\ \mathsf{Dummy}$$

Figure 2.8: Refined term reconstruction judgement

on the derivations of $\Gamma_1, x :: \sigma \mid \Gamma_2 \vdash_{outer} t$ and $\Gamma_1, x :: \sigma \mid \Gamma_2 \vdash_{inner} t$. Using properties 2 and 3, one may then prove property 1. Properties 4 and 5, expressing that $\vdash_{outer}$ is stronger than $\vdash_{inner}$ and that $\vdash_{outer}$ and $\vdash_{inner}$ characterize a subset of higher-order patterns, can be proved by induction on the derivation of $\vdash_{outer}$ and $\vdash_{inner}$.

We now show how the rules for reconstruction on terms given in Figure 2.5 can be extended to deal with terms containing term placeholders "_". To this end, we introduce the reconstruction judgement $\Gamma_1 \mid \Gamma_2 \vdash t_p \rhd_m (t,\ \tau,\ C)$. The rules characterizing this judgement, which have been obtained by modifying the rules from Figure 2.5, are shown in Figure 2.8. In analogy to the judgements $\vdash_{outer}$ and $\vdash_{inner}$, the refined reconstruction judgement involves two variable contexts Γ_1 and Γ_2, where Γ_1 again contains all variables bound by outer abstractions or abstractions which occur as an argument of a variable bound by an outer abstraction, while Γ_2 contains all other bound variables. Moreover, the refined reconstruction judgement is parameterized with a mode m, where $m \in \{inner, outer\}$. If the mode has no significance, as e.g. in the rules Var, Const, and Dummy, we just write $\Gamma_1 \mid \Gamma_2 \vdash t_p \rhd (t,\ \tau,\ C)$ instead of $\Gamma_1 \mid \Gamma_2 \vdash t_p \rhd_m (t,\ \tau,\ C)$. The rules Var and Const from Figure 2.8 are almost the same as their counterparts from Figure 2.5, the only difference being the representation of the variable context. The Dummy rule deals which the newly-introduced term placeholder "_", which is replaced by a unification variable depending only on the variables in Γ_2 during reconstruction. A variable $x :: \tau$ bound by a λ-abstraction is added to the context Γ_1 when reconstruction is in "*outer*" mode, and to Γ_2 when in "*inner*" mode by the rules Abs_{outer} and Abs_{inner}, respectively. Reconstruction of applications $(t_p\ u_p)$ is performed by the rules App_{outer} and App_{inner}. Reconstruction of the argument term u_p is performed in "*outer*" mode if the head of t_p is a variable x which is contained in the context Γ_1, otherwise (i.e. if the head is some other variable or a constant) in "*inner*" mode. Note that the mode is of no significance for the reconstruction of t_p. We again assume $(t_p\ u_p)$ to be in normal form, so the head of t_p may not be an abstraction. As in Figure 2.5, similar rules are required for the reconstruction of terms involving type placeholders. These have been omitted here in order to shorten the presentation. Since the term placeholder "_" is now part of the term language, the rules $\mathsf{Impl_p}$

and $\mathsf{AllE_p}$ from Figure 2.5 are no longer needed. Instead, the premises $\Gamma \vdash \varphi_p \rhd (\varphi,\ \tau,\ C)$ and $\Gamma \vdash t_p \rhd (t,\ \tau,\ D)$ of the rules Impl and AllE are replaced by $[] \mid \Gamma \vdash \varphi_p \rhd_{outer} (\varphi,\ \tau,\ C)$ and $[] \mid \Gamma \vdash t_p \rhd_{outer} (t,\ \tau,\ D)$, respectively, where $[]$ denotes the empty context.

The last missing piece is the compression algorithm for terms. Similar to the compression algorithm for proofs shown in Figure 2.6, this algorithm is given by a judgement $\Gamma_1 \mid \Gamma_2 \vdash t \gg_m t_p$ which, in analogy to the reconstruction judgement $\Gamma_1 \mid \Gamma_2 \vdash t_p \rhd_m (t,\ \tau,\ C)$, is parameterized by a mode $m \in \{inner, outer\}$ and involves two variable contexts Γ_1 and Γ_2. The inference rules characterizing this judgement are presented in Figure 2.9. The judgement $\Gamma_1 \mid \Gamma_2 \vdash t \gg_{inner} t_p$ replaces all terms t that do not contain any of the variables in Γ_1 by a placeholder, whereas the judgement $\Gamma_1 \mid \Gamma_2 \vdash t \gg_{outer} t_p$ only does so under the additional condition that t does not have a function type. When compressing a term which is an application $(a\ \overline{t})$ containing variables from Γ_1, its head a is retained and its parameter list $\overline{t}$ is replaced by the compressed parameter list $\overline{t_p}$, which is obtained by a recursive application of the compression algorithm.

As an example, consider the following term, which occurs in the proof of theorem *List.lexn-conv* from the Isabelle/HOL library [53]:

$$\begin{array}{l}
\lambda u.\ xys = a \mathbin{\#} list \longrightarrow (x,\ y) \in r \longrightarrow Suc\ (size\ list + size\ ys') = u \longrightarrow \\
\quad lexn\ r\ u = \\
\quad \{ua.\ (\lambda(xs,\ ys).\ size\ xs = u \wedge size\ ys = u \wedge \\
\qquad\quad (\exists\, xys\ x. \\
\qquad\qquad (\exists\, xs'.\ xs = xys\ @\ x \mathbin{\#} xs') \wedge \\
\qquad\qquad (\exists\, y.\ (\exists\, ys'.\ ys = xys\ @\ y \mathbin{\#} ys') \wedge (x,\ y) \in r)))\ ua\} \longrightarrow \\
\quad \neg\ (\exists\, xys\ xa.\ (\exists\, xs'a.\ list\ @\ x \mathbin{\#} xs' = xys\ @\ xa \mathbin{\#} xs'a) \wedge \\
\qquad\quad (\exists\, ya.\ (\exists\, ys'a.\ list\ @\ y \mathbin{\#} ys' = xys\ @\ ya \mathbin{\#} ys'a) \wedge (xa,\ ya) \in r)) \longrightarrow \\
\quad (a,\ a) \in r
\end{array}$$

The only variable bound by an *outer* abstraction is u, whereas all other variables such as ua, xs or ys are bound by *inner abstractions*. The compression algorithm may therefore omit all subterms not containing the variable u, which yields the term

$$\lambda u.\ \_ \longrightarrow \_ \longrightarrow \_ = u \longrightarrow lexn\ \_\ u = \{ua.\ (\lambda(xs,\ ys).\ \_ = u \wedge \_ = u \wedge \_)\ \_\} \longrightarrow \_$$

Compared to the original term consisting of 120 constructors, the compressed term only has 30 constructors, which is a compression ratio of 75%.

In order to integrate the term compression algorithm with the proof compression algorithm shown in Figure 2.6, we have to replace one of the rules for the compression of applications $(p \cdot t)$ by a rule involving the term compression judgement $\Gamma_1 \mid \Gamma_2 \vdash t \gg_{outer} t_p$. It should be noted that the term compression judgement is not immediately applicable to the compression of term arguments corresponding to top level predicate variables of type $\overline{\alpha} \Rightarrow \mathsf{prop}$, since this would require a more detailed analysis of both the polarity of the predicate variable and the structure of the formula denoted by the term. For example, in a proof $(c\ \cdots \cdot t)$, where $\Sigma(c) = \bigwedge \overline{a}.\ \varphi$ and t corresponds to a top level predicate variable $P \in \overline{a}$, t may only be omitted if P occurs only positively in φ and t contains no positive occurences of quantifiers $\bigwedge$, or P occurs only negatively in φ and t contains no negative occurrences of quantifiers, or if t contains no quantifiers at all. We therefore leave such term arguments unchanged. To this end, we introduce the additional representation recipe $\circ.r$ denoting an argument corresponding to such a predicate variable and also modify the function for generating representation recipes. The modified function, together with the modified rules for proof reconstruction, is shown in Figure 2.9 as well.

We are now ready to state the main result, relating the refined compression and reconstruction judgements:

$$\frac{\Gamma_1, x :: \tau \mid \Gamma_2 \vdash t \gg_{outer} t_p}{\Gamma_1 \mid \Gamma_2 \vdash (\lambda x :: \tau.\ t) \gg_{outer} (\lambda x :: \_.\ t_p)}$$

$$\frac{\mathit{Vars}(t) \cap \overline{V_{\Gamma_1}} = \emptyset \quad \Gamma_1, \Gamma_2 \not\vdash \sigma \Rightarrow \tau}{\Gamma_1 \mid \Gamma_2 \vdash t \gg_{outer} \_} \qquad \frac{\Gamma_1 \mid \Gamma_2 \Vdash \overline{t} \gg_{outer} \overline{t_p} \quad x \in \overline{V_{\Gamma_1}}}{\Gamma_1 \mid \Gamma_2 \vdash x\ \overline{t} \gg_{outer} x\ \overline{t_p}}$$

$$\frac{\Gamma_1 \mid \Gamma_2 \Vdash \overline{t} \gg_{inner} \overline{t_p} \quad a \in \overline{V_{\Gamma_2}} \cup \mathcal{C} \quad \mathit{Vars}(\overline{t}) \cap \overline{V_{\Gamma_1}} \neq \emptyset \text{ or } \Gamma_1, \Gamma_2 \vdash \sigma \Rightarrow \tau}{\Gamma_1 \mid \Gamma_2 \vdash a\ \overline{t} \gg_{outer} a\ \overline{t_p}}$$

$$\frac{}{\Gamma_1 \mid \Gamma_2 \Vdash [] \gg_{outer} []} \qquad \frac{\Gamma_1 \mid \Gamma_2 \vdash t \gg_{outer} t_p \quad \Gamma_1 \mid \Gamma_2 \Vdash \overline{t} \gg_{outer} \overline{t_p}}{\Gamma_1 \mid \Gamma_2 \Vdash t, \overline{t} \gg_{outer} t_p, \overline{t_p}}$$

$$\frac{\Gamma_1 \mid \Gamma_2, x :: \tau \vdash t \gg_{inner} t_p \quad \mathit{Vars}(\lambda x :: \tau.\ t) \cap \overline{V_{\Gamma_1}} \neq \emptyset}{\Gamma_1 \mid \Gamma_2 \vdash (\lambda x :: \tau.\ t) \gg_{inner} (\lambda x :: \_.\ t_p)}$$

$$\frac{\mathit{Vars}(t) \cap \overline{V_{\Gamma_1}} = \emptyset}{\Gamma_1 \mid \Gamma_2 \vdash t \gg_{inner} \_} \qquad \frac{\Gamma_1 \mid \Gamma_2 \Vdash \overline{t} \gg_{outer} \overline{t_p} \quad x \in \overline{V_{\Gamma_1}}}{\Gamma_1 \mid \Gamma_2 \vdash x\ \overline{t} \gg_{inner} x\ \overline{t_p}}$$

$$\frac{\Gamma_1 \mid \Gamma_2 \Vdash \overline{t} \gg_{inner} \overline{t_p} \quad a \in \overline{V_{\Gamma_2}} \cup \mathcal{C} \quad \mathit{Vars}(\overline{t}) \cap \overline{V_{\Gamma_1}} \neq \emptyset}{\Gamma_1 \mid \Gamma_2 \vdash a\ \overline{t} \gg_{inner} a\ \overline{t_p}}$$

$$\frac{}{\Gamma_1 \mid \Gamma_2 \Vdash [] \gg_{inner} []} \qquad \frac{\Gamma_1 \mid \Gamma_2 \vdash t \gg_{inner} t_p \quad \Gamma_1 \mid \Gamma_2 \Vdash \overline{t} \gg_{inner} \overline{t_p}}{\Gamma_1 \mid \Gamma_2 \Vdash t, \overline{t} \gg_{inner} t_p, \overline{t_p}}$$

$$\frac{\Gamma \vdash p \gg (p_p,\ -.r)}{\Gamma \vdash p \cdot t \gg (p_p \cdot \_,\ r)} \qquad \frac{\Gamma \vdash p \gg (p_p,\ \circ.r)}{\Gamma \vdash p \cdot t \gg (p_p \cdot t,\ r)} \qquad \frac{\Gamma \vdash p \gg (p_p,\ \bullet.r) \quad [] \mid \Gamma \vdash t \gg_{outer} t_p}{\Gamma \vdash p \cdot t \gg (p_p \cdot t_p,\ r)}$$

$$r \;=\; -.r \mid \circ.r \mid \bullet.r \mid []$$

$$\begin{aligned}
\mathsf{recipe}_{\mathcal{A},\varphi}\ [] &= [] \\
\mathsf{recipe}_{\mathcal{A},\varphi}\ (a, \overline{a}) &= \begin{cases} \circ.\mathsf{recipe}_{\mathcal{A},\varphi}\ \overline{a} & \text{if } a \in \mathsf{propv}_{\mathcal{A}}\ (\varphi) \\ \bullet.\mathsf{recipe}_{\mathcal{A},\varphi}\ \overline{a} & \text{if } a \in \mathsf{unsafe}^{+}_{\mathcal{A}}\ \{\}\ (\varphi) \\ -.\mathsf{recipe}_{\mathcal{A},\varphi}\ \overline{a} & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 2.9: Refined compression of terms

Theorem 2.5 (Refined compression)

1. If t is ground, $\Gamma_1 \mid \Gamma_2 \vdash t \gg_{inner} t_p$ and $\Gamma_1 \mid \Gamma_2 \vdash t_p \rhd_{inner} (t',\ \tau,\ C)$, then $\Gamma_1 \mid \Gamma_2 \vdash_{inner} t'$
2. If t is ground, $\Gamma_1 \mid \Gamma_2 \vdash t \gg_{outer} t_p$ and $\Gamma_1 \mid \Gamma_2 \vdash t_p \rhd_{outer} (t',\ \tau,\ C)$, then $\Gamma_1 \mid \Gamma_2 \vdash_{outer} t'$
3. If $\Gamma \vdash p : \varphi$ and $\Gamma \vdash p \gg (p_p,\ r)$ and $\Gamma \vdash p_p \rhd (p',\ \varphi',\ C)$, then $C \cup \{\varphi =^? \varphi'\}$ is solvable using pattern unification.

2.4.5 A bidirectional compression algorithm

As has been noted in §2.4.3, a parameter of an inference rule r, where $\Sigma(r) = \bigwedge \overline{a}.\ \varphi$, may be omitted if it has only *strict* occurrences in φ, since this preserves solvability of the constraints generated during reconstruction using pattern unification. It is therefore tempting to ask if this also holds for parameters having both strict and non-strict occurrences, as in the rules

$$\begin{array}{l} exI : \bigwedge P\ x.\ \ \mathsf{Tr}\ (\overbrace{P\ x}^{\text{non-strict}}) \Longrightarrow \mathsf{Tr}\ (\exists x.\ \overbrace{P}^{\text{strict}}\ x) \\ spec : \bigwedge P\ x.\ \ \mathsf{Tr}\ (\forall x.\ \underbrace{P}_{\text{strict}}\ x) \Longrightarrow \mathsf{Tr}\ (\underbrace{P\ x}_{\text{non-strict}}) \end{array}$$

where the parameter P has both a strict and a non-strict occurrence, whereas the parameter x has only a non-strict occurrence. The following example shows that this is *not* the case in general. For the purpose of the example, assume we already have a proof of the theorem

$$example : \bigwedge P.\ \ \mathsf{Tr}\ (\forall x.\ \forall y.\ P\ x\ y \longrightarrow P\ x\ y)$$

where the parameter P occurring in the proposition of *example* has only positive occurrences. Now consider the partial proof

$$spec \cdot \_ \cdot c \bullet (spec \cdot \_ \cdot c \bullet (example \cdot \_))$$

where c is some constant. According to §2.4.3, we may clearly omit the parameter P of the rule *example*, since it has only positive occurrences. However, if we go even further and also omit the parameter P of *spec*, we run into a problem. To see this, consider the reconstruction problem

$$spec \cdot ?P_1 \cdot c \bullet (spec \cdot ?P_2 \cdot c \bullet (example \cdot ?P_3))\ \ :\ \ \mathsf{Tr}\ (P\ c\ c \longrightarrow P\ c\ c)$$

which generates the following set of constraints:

$$\{?P_1\ \ c =^? P\ c\ c \longrightarrow P\ c\ c,\ ?P_2\ \ c =^? (\forall x.\ ?P_1\ \ x),\ (\forall x.\ \forall y.\ ?P_3\ \ x\ y \longrightarrow ?P_3\ \ x\ y) =^? (\forall x.\ ?P_2\ \ x)\}$$

These constraints are not solvable using pattern unification, and although we can deduce the substitution

$$\{?P_1 \mapsto (\lambda y.\ ?P_3\ \ c\ y \longrightarrow ?P_3\ \ c\ y),\ ?P_2 \mapsto (\lambda x.\ \forall y.\ ?P_3\ \ x\ y \longrightarrow ?P_3\ \ x\ y)\}$$

which simplifies the above constraint set to

$$\{?P_3\ \ c\ c \longrightarrow ?P_3\ \ c\ c =^? P\ c\ c \longrightarrow P\ c\ c\}$$

no unique most general solution exists. For example, some of the solutions of the above constraint set are

$$\left\{\begin{array}{l} ?P_1 \mapsto \lambda y.\ P\ c\ y \longrightarrow P\ c\ y \\ ?P_2 \mapsto \lambda x.\ \forall y.\ P\ x\ y \longrightarrow P\ x\ y \\ ?P_3 \mapsto P \end{array}\right\},\ \left\{\begin{array}{l} ?P_1 \mapsto \lambda y.\ P\ c\ c \longrightarrow P\ c\ c \\ ?P_2 \mapsto \lambda x.\ \forall y.\ P\ c\ c \longrightarrow P\ c\ c \\ ?P_3 \mapsto \lambda x\ y.\ P\ c\ c \end{array}\right\},\ \ldots$$

In contrast to the above example, consider the partial proof

$$\mathit{impI} \cdot \_ \cdot \_ \bullet (\boldsymbol{\lambda} H : \_.\ \mathit{exI} \cdot \_ \cdot c \bullet (\mathit{spec} \cdot \_ \cdot c \bullet H))$$

where the parameter P of *exI* has been omitted. Although P has both a strict and a non-strict occurrence, reconstruction of this proof is unproblematic. The reconstruction problem

$$\mathit{impI} \cdot ?A \cdot ?B \bullet (\boldsymbol{\lambda} H : ?C.\ \mathit{exI} \cdot ?P_1 \cdot c \bullet (\mathit{spec} \cdot ?P_2 \cdot c \bullet H))\ :\ \mathsf{Tr}\ ((\forall x.\ P\ x) \longrightarrow (\exists x.\ P\ x))$$

generates the constraints

$$C = \left\{\begin{array}{l} ?A \longrightarrow ?B =^? (\forall x.\ P\ x) \longrightarrow (\exists x.\ P\ x),\ ?C \Longrightarrow \mathsf{Tr}\ (\exists x.\ ?P_1\ x) =^? \mathsf{Tr}\ ?A \Longrightarrow \mathsf{Tr}\ ?B \\ ?C =^? \mathsf{Tr}\ (\forall x.\ ?P_2\ x),\ ?P_2\ c =^? ?P_1\ c \end{array}\right\}$$

which are solvable using pattern unification, since

$$\begin{array}{l} (C,\ \{\}) \longrightarrow^*_{\mathcal{S}} \\ \left(\left\{\begin{array}{l} ?C \Longrightarrow \mathsf{Tr}\ (\exists x.\ ?P_1\ x) =^? \mathsf{Tr}\ (\forall x.\ P\ x) \Longrightarrow \mathsf{Tr}\ (\exists x.\ P\ x) \\ ?C =^? \mathsf{Tr}\ (\forall x.\ ?P_2\ x),\ ?P_2\ c =^? ?P_1\ c \end{array}\right\},\ \left\{\begin{array}{l} ?A \mapsto (\forall x.\ P\ x) \\ ?B \mapsto (\exists x.\ P\ x) \end{array}\right\}\right) \longrightarrow^*_{\mathcal{S}} \\ \left(\left\{\begin{array}{l} \mathsf{Tr}\ (\forall x.\ P\ x) =^? \mathsf{Tr}\ (\forall x.\ ?P_2\ x) \\ ?P_2\ c =^? P\ c \end{array}\right\},\ \left\{\begin{array}{l} ?A \mapsto (\forall x.\ P\ x),\ ?B \mapsto (\exists x.\ P\ x) \\ ?C \mapsto \mathsf{Tr}\ (\forall x.\ P\ x),\ ?P_1 \mapsto P \end{array}\right\}\right) \longrightarrow^*_{\mathcal{S}} \\ (\{\},\ \{?A \mapsto (\forall x.\ P\ x),\ ?B \mapsto (\exists x.\ P\ x),\ ?C \mapsto \mathsf{Tr}\ (\forall x.\ P\ x),\ ?P_1 \mapsto P,\ ?P_2 \mapsto P\}) \end{array}$$

The reason why reconstruction fails in the first example, whereas it succeeds in the second example, is that the rules *exI* and *spec* only admit specific directions of *dataflow*. The rule *exI* requires the dataflow to be bottom-up: when given a ground term to match with the conclusion $(\exists x.\ ?P\ x)$ of $(\mathit{exI} \cdot ?P \cdot t)$, a ground instantiation for $?P$ can be computed, which, assuming that t is ground, also instantiates the premise $(?P\ t)$ to a ground term. However, if we are given a ground term to match with the premise $(?P\ t)$, we can not uniquely determine an instantiation for $?P$. In contrast, the rule *spec* requires the dataflow to be top-down: when given a ground term to match with the premise $(\forall x.\ ?P\ x)$ of $(\mathit{spec} \cdot ?P \cdot \_)$, we can compute a ground instantiation for $?P$, which also instantiates the conclusion $(?P\ t)$. However, when we are just given a ground term to match with the conclusion, we again cannot uniquely determine $?P$. In the second example, the direction of dataflow is exactly as it should be: working bottom-up starting from the specified result proposition $\mathsf{Tr}\ ((\forall x.\ P\ x) \longrightarrow (\exists x.\ P\ x))$, we can first reconstruct the conclusion of $(\mathit{exI} \cdot ?P_1 \cdot c)$ and hence also its premise, which then allows us to reconstruct the conclusion of $(\mathit{spec} \cdot ?P_2 \cdot c)$. Unfortunately, in the first example, the dataflow is in the "wrong" direction. Although we are given the result proposition $\mathsf{Tr}\ (P\ c\ c \longrightarrow P\ c\ c)$, this does not help very much, since we cannot use it to determine a ground instantiation for the variable $?P_1$ in the conclusion of $(\mathit{spec} \cdot ?P_1 \cdot c)$. A way out of this problem is to keep the argument of *example* in the proof, instead of replacing it by a placeholder, which then gives rise to the unification variable $?P_3$ during reconstruction. Working top-down, which is the preferred direction of dataflow for the rule *spec*, we could thus determine $?P_2$ and hence also $?P_1$.

The idea behind the bidirectional compression algorithm which will be developed in this section is to assign dataflow annotations to each inference rule. This is similar to the concept of *mode analysis*, which will be introduced in §6.3.1 for the purpose of translating logic programs into functional programs. The set of possible dataflow annotations is specified by the following grammar:

$$F_1, F_2, F \;=\; \upharpoonleft \;|\; \downharpoonleft \;|\; F_1 \Longrightarrow F_2$$

Intuitively, $\upharpoonleft$ means that the direction of dataflow is bottom-up, whereas $\downharpoonleft$ means that it is top-down. For example, the preferred direction of dataflow for the rule *exI* described above is denoted by the annotation $\upharpoonleft \Longrightarrow \upharpoonleft$, i.e. information flows from the conclusion to the premise, whereas the annotation for the rule *spec* is $\downharpoonleft \Longrightarrow \downharpoonleft$, i.e. information flows from the premise to the conclusion. These dataflow annotations are then used to guide the compression algorithm. For each dataflow annotation of a specific rule, we can determine a set of parameters, which need to be retained during compression, in order to allow for a successful reconstruction of the proof. For example, given the above annotations for the rules *exI* and *spec*, the term argument corresponding to the parameter x always has to be retained in the proof. Formally, the set of parameters of an inference rule r, for which an instantiation can be inferred, given a particular dataflow direction F, is defined as follows:

Definition 2.7 (Inferrable parameters) Let r be an inference rule (proof constant) with $\Sigma(r) = \bigwedge a_1 \ldots a_n.\ \varphi$, where $\mathcal{A} = \{a_1, \ldots, a_n\}$ is the set of its parameters. Then the set $\mathsf{inferrable}_{\mathcal{A}}\ F\ \varphi$ of *inferrable parameters* of r wrt. the dataflow annotation F is given by the following equations:

$$\begin{array}{lcl}
\mathsf{inferrable}^{+}_{\mathcal{A}}\ \upharpoonleft\ \varphi & = & \mathsf{strict}_{\mathcal{A}}\ \varphi \\
\mathsf{inferrable}^{-}_{\mathcal{A}}\ \upharpoonleft\ \varphi & = & \{\} \\
\mathsf{inferrable}^{+}_{\mathcal{A}}\ \downharpoonleft\ \varphi & = & \{\} \\
\mathsf{inferrable}^{-}_{\mathcal{A}}\ \downharpoonleft\ \varphi & = & \mathsf{strict}_{\mathcal{A}}\ \varphi \\
\mathsf{inferrable}^{\pi}_{\mathcal{A}}\ F\ (\bigwedge x.\ \varphi) & = & \mathsf{inferrable}^{\pi}_{\mathcal{A}}\ F\ \varphi \\
\mathsf{inferrable}^{\pi}_{\mathcal{A}}\ (F_1 \Longrightarrow F_2)\ (\varphi_1 \Longrightarrow \varphi_2) & = & \mathsf{inferrable}^{-\pi}_{\mathcal{A}}\ F_1\ \varphi_1 \cup \mathsf{inferrable}^{\pi}_{\mathcal{A}}\ F_2\ \varphi_2
\end{array}$$

Intuitively, if $\upharpoonleft$ occurs *positively* in a dataflow annotation, this means that we may assume that we have a ground term to match with the corresponding subformula φ, and thus can infer a ground instantiation for all parameters having a *strict* occurrence in φ. The same holds for *negative* occurrences of $\downharpoonleft$ in a dataflow annotation. In contrast, if $\upharpoonleft$ occurs *negatively*, or if $\downharpoonleft$ occurs *positively* in a dataflow annotation, we have to ensure that it is possible to obtain ground terms for all parameters occurring in the corresponding subformula φ. Figure 2.10 shows a list of dataflow annotations for the standard inference rules of Isabelle/HOL. Note that for each rule, there are usually several possible annotations. These may differ in the number of non-inferrable parameters, and the most common annotation is listed first. Not all dataflow annotations are meaningful. For example, the annotation

$$\downharpoonleft \Longrightarrow (\downharpoonleft \Longrightarrow \downharpoonleft) \Longrightarrow (\downharpoonleft \Longrightarrow \downharpoonleft) \Longrightarrow \downharpoonleft$$

for the rule

$$\mathit{disjE} : \bigwedge P\ Q\ R.\ P \vee Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$$

rule	annotation	non-inferrable parameters
allI	$\uparrow \Longrightarrow \uparrow$	$\{\}$
	$\downarrow \Longrightarrow \downarrow$	$\{\}$
	$\uparrow \Longrightarrow \downarrow$	$\{P\}$
spec	$\downarrow \Longrightarrow \downarrow$	$\{x\}$
	$\uparrow \Longrightarrow \downarrow$	$\{P, x\}$
exI	$\uparrow \Longrightarrow \uparrow$	$\{x\}$
	$\uparrow \Longrightarrow \downarrow$	$\{P, x\}$
exE	$\downarrow \Longrightarrow (\downarrow \Longrightarrow \uparrow) \Longrightarrow \uparrow$	$\{\}$
	$\downarrow \Longrightarrow (\downarrow \Longrightarrow \downarrow) \Longrightarrow \downarrow$	$\{\}$
	$\downarrow \Longrightarrow (\downarrow \Longrightarrow \uparrow) \Longrightarrow \downarrow$	$\{Q\}$
	$\vdots$	$\vdots$
impI	$(\downarrow \Longrightarrow \uparrow) \Longrightarrow \uparrow$	$\{\}$
	$(\downarrow \Longrightarrow \downarrow) \Longrightarrow \downarrow$	$\{P\}$
	$\vdots$	$\vdots$
mp	$\downarrow \Longrightarrow \uparrow \Longrightarrow \downarrow$	$\{\}$
	$\uparrow \Longrightarrow \downarrow \Longrightarrow \uparrow$	$\{\}$
	$\uparrow \Longrightarrow \uparrow \Longrightarrow \uparrow$	$\{P\}$
	$\vdots$	$\vdots$
conjI	$\uparrow \Longrightarrow \uparrow \Longrightarrow \uparrow$	$\{\}$
	$\downarrow \Longrightarrow \downarrow \Longrightarrow \downarrow$	$\{\}$
	$\downarrow \Longrightarrow \uparrow \Longrightarrow \downarrow$	$\{Q\}$
	$\vdots$	$\vdots$
conjunct1	$\downarrow \Longrightarrow \downarrow$	$\{\}$
	$\uparrow \Longrightarrow \uparrow$	$\{Q\}$
	$\uparrow \Longrightarrow \downarrow$	$\{P, Q\}$
conjunct2	$\downarrow \Longrightarrow \downarrow$	$\{\}$
	$\uparrow \Longrightarrow \uparrow$	$\{P\}$
	$\uparrow \Longrightarrow \downarrow$	$\{P, Q\}$
disjI1	$\uparrow \Longrightarrow \uparrow$	$\{\}$
	$\downarrow \Longrightarrow \downarrow$	$\{Q\}$
	$\uparrow \Longrightarrow \downarrow$	$\{P, Q\}$
disjI2	$\uparrow \Longrightarrow \uparrow$	$\{\}$
	$\downarrow \Longrightarrow \downarrow$	$\{P\}$
	$\uparrow \Longrightarrow \downarrow$	$\{P, Q\}$
disjE	$\downarrow \Longrightarrow (\downarrow \Longrightarrow \uparrow) \Longrightarrow (\downarrow \Longrightarrow \uparrow) \Longrightarrow \uparrow$	$\{\}$
	$\downarrow \Longrightarrow (\downarrow \Longrightarrow \downarrow) \Longrightarrow (\downarrow \Longrightarrow \uparrow) \Longrightarrow \downarrow$	$\{\}$
	$\downarrow \Longrightarrow (\downarrow \Longrightarrow \uparrow) \Longrightarrow (\downarrow \Longrightarrow \downarrow) \Longrightarrow \downarrow$	$\{\}$
	$\downarrow \Longrightarrow (\downarrow \Longrightarrow \uparrow) \Longrightarrow (\downarrow \Longrightarrow \uparrow) \Longrightarrow \downarrow$	$\{R\}$
	$\vdots$	$\vdots$

Figure 2.10: Dataflow annotations for standard inference rules

does not make sense, since if we have already synthesized R from the proof of $(P \Longrightarrow R)$, there is no point in synthesizing it again from the proof of $(Q \Longrightarrow R)$. Instead, we may already use the knowledge of R when reconstructing the proof of $(Q \Longrightarrow R)$, which is expressed by the annotation

$$\downarrow \Longrightarrow (\downarrow \Longrightarrow \downarrow) \Longrightarrow (\downarrow \Longrightarrow \uparrow) \Longrightarrow \downarrow$$

Moreover, neither $\downarrow \Longrightarrow \uparrow$ nor $\downarrow \Longrightarrow \downarrow$ would make sense as an annotation for *exI*, since the knowledge of a ground term to match against the premise $P\ x$ would not help in inferring any of the parameters, since none of them has a strict occurrence in the premise. More precisely, a dataflow annotation is considered meaningful if it is not *redundant* in the sense of the following definition.

Definition 2.8 (Redundant annotation) Let r be an inference rule (proof constant) with $\Sigma(r) = \bigwedge a_1 \ldots a_n.\ \varphi$, where $\mathcal{A} = \{a_1, \ldots, a_n\}$ is the set of its parameters. A dataflow annotation F for r is called *redundant*, iff $\mathsf{redundant}^+_{\mathcal{A}}\ F\ \varphi$, where

$$\begin{array}{lcl}
\mathsf{redundant}^+_{\mathcal{A}}\ \uparrow\ \varphi & = & (\mathsf{strict}_{\mathcal{A}}\ \varphi = \emptyset) \\
\mathsf{redundant}^-_{\mathcal{A}}\ \uparrow\ \varphi & = & \mathsf{False} \\
\mathsf{redundant}^+_{\mathcal{A}}\ \downarrow\ \varphi & = & \mathsf{False} \\
\mathsf{redundant}^-_{\mathcal{A}}\ \downarrow\ \varphi & = & (\mathsf{strict}_{\mathcal{A}}\ \varphi = \emptyset) \\
\mathsf{redundant}^\pi_{\mathcal{A}}\ F\ (\bigwedge x.\ \varphi) & = & \mathsf{redundant}^\pi_{\mathcal{A}}\ F\ \varphi \\
\mathsf{redundant}^\pi_{\mathcal{A}}\ (F_1 \Longrightarrow F_2)\ (\varphi_1 \longrightarrow \varphi_2) & = & \mathsf{redundant}^{-\pi}_{\mathcal{A}}\ F_1\ \varphi_1 \vee \mathsf{redundant}^\pi_{\mathcal{A}}\ F_2\ \varphi_2\ \vee \\
 & & \emptyset \neq \mathsf{inferrable}^{-\pi}_{\mathcal{A}}\ F_1\ \varphi_1 \subseteq \mathsf{inferrable}^\pi_{\mathcal{A}}\ F_2\ \varphi_2\ \vee \\
 & & \mathsf{inferrable}^{-\pi}_{\mathcal{A}}\ F_1\ \varphi_1 \supseteq \mathsf{inferrable}^\pi_{\mathcal{A}}\ F_2\ \varphi_2 \neq \emptyset
\end{array}$$

In the sequel, we will assume all dataflow annotations to be non-redundant.

The bidirectional compression algorithm is given by two judgements $\Gamma \vdash p : F \gg p_p$ and $\Gamma \vdash p \gg (p_p,\ r,\ F)$, which are characterized by the inference rules in Figure 2.11. The first of the two judgements handles the compression of proofs whose outermost constructor is an abstraction, whereas the second one is intended for compressing applications. As in the static compression algorithm from §2.4.3, representation recipes are only needed for the compression of applications. During compression, we will sometimes have to mediate between different directions of dataflow. For example, a proof which can synthesize its proposition may appear in any context expecting a proof which can be checked against a given proposition. In other words, a proof in which the direction of dataflow is top-down (indicated by $\downarrow$) may appear in any context in which the dataflow is bottom-up (indicated by $\uparrow$). To this end, we introduce a *subsumption relation* $\preceq$ on dataflow annotations, together with a rule Sub for the compression judgement, which allows a proof with dataflow F' to appear in a context with dataflow F, provided that $F \preceq F'$. As is customary for subtyping relations, $\preceq$ is *contravariant* in the first argument F, and *covariant* in the second argument G of $F \Longrightarrow G$. Moreover, $\preceq$ also allows to convert between dataflow annotations with different degrees of granularity. For example, $\uparrow$ and $\downarrow$ are equivalent to $\downarrow \Longrightarrow \uparrow$ and $\uparrow \Longrightarrow \downarrow$, respectively. As an invariant of the compression algorithm, we require that the propositions corresponding to hypotheses in the context Γ can always be synthesized. Therefore, when compressing an abstraction of the form $(\boldsymbol{\lambda} h : \varphi.\ p)$ using the rules Abs_1 and Abs_2, the dataflow must either be completely bottom-up ($\uparrow$), or such that it at least allows a synthesis of the argument proposition φ, which can be captured by the annotation $\downarrow \Longrightarrow F$. Consequently, the Hyp rule for processing hypotheses assumes that the direction of dataflow is top-down, which means that information flows from the hypotheses to

$$\frac{\Gamma, h : \varphi \vdash p : F \gg p_p}{\Gamma \vdash (\boldsymbol{\lambda} h : \varphi.\ p) : \downarrow \Longrightarrow F \gg (\boldsymbol{\lambda} h : \_.\ p_p)}\ \mathsf{Abs}_1 \qquad \frac{\Gamma, h : \varphi \vdash p : \uparrow \gg p_p}{\Gamma \vdash (\boldsymbol{\lambda} h : \varphi.\ p) : \uparrow \gg (\boldsymbol{\lambda} h : \_.\ p_p)}\ \mathsf{Abs}_2$$

$$\frac{\Gamma \vdash p : F \gg p_p}{\Gamma \vdash (\boldsymbol{\lambda} x :: \tau.\ p) : F \gg (\boldsymbol{\lambda} x :: \_.\ p_p)}\ \mathsf{Abs}_3$$

$$\frac{F \preceq F' \quad \Gamma \vdash p \gg (p_p,\ r,\ F')}{\Gamma \vdash p : F \gg p_p}\ \mathsf{Sub}$$

$$\frac{\Sigma(c) = \bigwedge \overline{a}.\ \varphi \quad F \in \mathcal{F}(c)}{\Gamma \vdash c_{\{\overline{\alpha} \mapsto \overline{\tau}\}} \gg (c_{\{\overline{\alpha} \mapsto \_\}},\ \mathsf{recipe}_{\overline{a},\varphi,F}\ \overline{a},\ F)}\ \mathsf{Const} \qquad \frac{}{\Gamma \vdash h \gg (h,\ [],\ \downarrow)}\ \mathsf{Hyp}$$

$$\frac{\Gamma \vdash p \gg (p_p,\ -.r,\ F)}{\Gamma \vdash p \cdot t \gg (p_p \cdot \_,\ r,\ F)}\ \mathsf{App}^{-} \qquad \frac{\Gamma \vdash p \gg (p_p,\ \bullet.r,\ F)}{\Gamma \vdash p \cdot t \gg (p_p \cdot t,\ r,\ F)}\ \mathsf{App}^{\bullet} \qquad \frac{\Gamma \vdash p \gg (p_p,\ [],\ \downarrow)}{\Gamma \vdash p \cdot t \gg (p_p \cdot t,\ r,\ \downarrow)}\ \mathsf{App}^{[]}$$

$$\frac{\Gamma \vdash p \gg (p_p,\ r,\ F \Longrightarrow F') \quad \Gamma \vdash q : F \gg q_p}{\Gamma \vdash p \bullet q \gg (p_p \bullet q_p,\ [],\ F')} \qquad \frac{\Gamma \vdash p \gg (p_p,\ r,\ \downarrow) \quad \Gamma \vdash q : \uparrow \gg q_p}{\Gamma \vdash p \bullet q \gg (p_p \bullet q_p,\ [],\ \downarrow)}$$

$$\uparrow \preceq \uparrow \qquad \uparrow \preceq \downarrow \qquad \downarrow \preceq \downarrow$$

$$\frac{F' \preceq F \quad G \preceq G'}{F \Longrightarrow G \preceq F' \Longrightarrow G'} \qquad \frac{F \preceq \downarrow \quad \uparrow \preceq G}{\uparrow \preceq F \Longrightarrow G} \qquad \frac{F \preceq \uparrow \quad \downarrow \preceq G}{\downarrow \preceq F \Longrightarrow G} \qquad \frac{\downarrow \preceq F \quad G \preceq \uparrow}{F \Longrightarrow G \preceq \uparrow} \qquad \frac{\uparrow \preceq F \quad G \preceq \downarrow}{F \Longrightarrow G \preceq \downarrow}$$

$$\begin{array}{lcl}
\mathsf{recipe}_{\mathcal{A},\varphi,F}\ [] & = & [] \\
\mathsf{recipe}_{\mathcal{A},\varphi,F}\ (a, \overline{a}) & = & \begin{cases} \bullet.\mathsf{recipe}_{\mathcal{A},\varphi,F}\ \overline{a} & \text{if } a \notin \mathsf{inferrable}^{+}_{\mathcal{A}}\ F\ \varphi \\ -.\mathsf{recipe}_{\mathcal{A},\varphi,F}\ \overline{a} & \text{otherwise} \end{cases}
\end{array}$$

Figure 2.11: Bidirectional compression

the root of the proof. Note that the above restriction on dataflow annotations for abstractions rules out annotations such as

$$\uparrow \Longrightarrow (\uparrow \Longrightarrow \uparrow) \Longrightarrow (\uparrow \Longrightarrow \uparrow) \Longrightarrow \uparrow$$

for the rule *disjE*. Similar to the static compression algorithm described in §2.4.3, representation recipes are generated by the rule Const for processing proof constants. This rule allows to choose a suitable dataflow annotation F from the set $\mathcal{F}(c)$ of possible dataflow annotations for the constant c. The generated representation recipe is a list containing all variables which are not inferrable wrt. the chosen dataflow annotation F. The rules for compressing applications of the form $p \cdot t$ are quite similar to their counterparts in §2.4.3. When compressing an application of the form $p \bullet q$, where p has the dataflow $F \Longrightarrow F'$, the proof q is compressed using the dataflow F, and F' is returned as the resulting dataflow annotation of the whole proof. Alternatively, if the dataflow corresponding to p is completely top-down ($\downarrow$), which means that we can synthesize both the premise φ and the conclusion ψ of the proposition $\varphi \Longrightarrow \psi$ proved by p, we may compress q in bottom-up mode ($\uparrow$), and the dataflow for the whole proof is again top-down ($\downarrow$).

Similar to the previous compression strategy, the interplay between compression and reconstruction can be characterized as follows:

Theorem 2.6 (Bidirectional compression) If $\Gamma \vdash p : \varphi$ and $\Gamma \vdash p : \uparrow \gg p_p$ and $\Gamma \vdash p_p \rhd (p',\ \varphi',\ C)$, then $C \cup \{\varphi =^? \varphi'\}$ is solvable using pattern unification.

It should be noted that the system of inference rules presented in Figure 2.11 does not immediately yield an executable program, since it still contains several degrees of freedom. First of all, we have to make more precise which dataflow annotation $F \in \mathcal{F}(c)$ to choose in the Const rule. As a rule of thumb, one should always try to choose a dataflow annotation with the smallest possible number of non-inferrable parameters. Moreover, when selecting the dataflow annonation, we also have to take into account the direction of dataflow expected by the context. For example, if the context requires the proof $c \cdot \bar{t} \cdot \cdots$ to synthesize its proposition, we have to choose a dataflow annotation of the form $\cdots \Longrightarrow \downarrow$ for c. If the context expects a proof which can be checked against a given proposition, it is usually more advantageous to choose a dataflow annotation of the form $\cdots \Longrightarrow \uparrow$, although, according to the Sub rule, $\cdots \Longrightarrow \downarrow$ would be acceptable as well. For example, the annotations

$$\downarrow \Longrightarrow (\downarrow \Longrightarrow \uparrow) \Longrightarrow (\downarrow \Longrightarrow \uparrow) \Longrightarrow \uparrow$$
$$\downarrow \Longrightarrow (\downarrow \Longrightarrow \downarrow) \Longrightarrow (\downarrow \Longrightarrow \uparrow) \Longrightarrow \downarrow$$

for *disjE* both allow all parameters to be inferred, but when compressing a proof of the form

$$\mathit{disjE} \cdot P \cdot Q \cdot R \bullet \mathit{prf}_1 \bullet \mathit{prf}_2 \bullet \mathit{prf}_3$$

the first annotation is likely to allow for more terms to be omitted in the subproof prf_2, since the annotation $\downarrow \Longrightarrow \uparrow$ (as opposed to $\downarrow \Longrightarrow \downarrow$) corresponding to prf_2 indicates that more information can be propagated upwards to prf_2, where it may help in reconstructing omitted arguments. Thus, in an actual implementation, the rules Const and Sub need to be more tightly integrated.

An important property of the bidirectional compression algorithm is that the reconstruction algorithm can infer *ground terms* for all placeholders in the compressed term. Thus, there may be cases where the compression achieved by this algorithm is less optimal than the one achieved by the static compression algorithm, which only requires that *pattern terms* can be inferred. For example, the proof

$$\mathit{disjE} \cdot \mathit{True} \cdot P \cdot \mathit{True} \bullet (\mathit{disjI1} \cdot \mathit{True} \cdot P \bullet \mathit{TrueI}) \bullet (\boldsymbol{\lambda} H : \mathit{True}.\ H) \bullet (\boldsymbol{\lambda} H : P.\ \mathit{TrueI})$$

is compressed to

$$\mathit{disjE} \cdot \_ \cdot \_ \cdot \_ \bullet (\mathit{disjI1} \cdot \_ \cdot P \bullet \mathit{TrueI}) \bullet (\boldsymbol{\lambda} H : \_.\ H) \bullet (\boldsymbol{\lambda} H : \_.\ \mathit{TrueI})$$

by the bidirectional compression algorithm. While the static algorithm would also omit P, the bidirectional algorithm cannot do so, since there is no way to fully reconstruct it from the context.

2.4.6 Practical results

We now conclude the presentation of the various compression algorithms given in the previous sections with an analysis of their performance. To this end, we have tested the compression algorithms on all (roughly 4000) theorems of the Isabelle/HOL library [53]. The detailed results of this test for the 30 largest proofs in the library are shown in the table in Figure 2.12. The first three columns of the table show the name of the theorem whose proof is considered, the number of terms appearing in the uncompressed (full) proof, as well as the total size of these terms. The remaining eight columns show the results of four different compression algorithms, namely the algorithm currently used in Isabelle, the static compression algorithm from §2.4.3, the refined compression algorithm from §2.4.4, as well as the bidirectional compression algorithm presented in §2.4.5. For each algorithm, there are two columns, showing the total size of the terms occurring in the compressed proof, as well as the compression ratio (in %) compared with the original proof. The diagram below the table in Figure 2.12 shows the correlation between the proof size and the compression ratio.

As expected, the algorithm with the poorest performance is the static algorithm, which only achieves a compression ratio of about 50-60%. The refined compression algorithm is already much better, yielding a compression ratio of about 80%. An even better compression ratio of about 90% can be achieved using the bidirectional compression algorithm. The compression algorithm currently used in Isabelle is essentially a static algorithm similar to the one described in §2.4.3, but with a specific optimization for equational proofs, borrowing some ideas from §2.4.5. This optimization is based on the observation that equational proofs mainly consist of applications of the congruence rule

$$\mathit{comb} : \bigwedge f\ g\ x\ y.\ f \equiv g \Longrightarrow x \equiv y \Longrightarrow f\ x \equiv g\ y$$

Using the static algorithm, none of the parameters f, g, x and y may be omitted, because each of them has a *non-strict occurrence* in the conclusion of the above rule. However, if the direction of dataflow in the proof is top-down, which can be expressed by the dataflow annotation $\downarrow \Longrightarrow \downarrow \Longrightarrow \downarrow$, we can actually omit all parameters. This is due to the fact that f, g, x and y have only *strict occurrences* in the premises of *comb*. Hence, when given ground terms to match against the premises $f \equiv g$ and $x \equiv y$, all parameters can be reconstructed. Therefore, when compressing a proof of the form

$$\mathit{comb} \cdot f \cdot g \cdot x \cdot y \bullet \mathit{prf}_1 \bullet \mathit{prf}_2$$

the following strategy is used: If prf_1 is again a proof starting with the *comb* rule, or a transitivity rule applied to two proofs starting with *comb*, we can rely on the fact that we can reconstruct the proposition corresponding to this proof, and can therefore omit the parameters f and g. In fact, it actually suffices if we can just reconstruct a pattern term whose head is a constant, since this is already sufficient to guarantee that also $f\ x$ and $g\ x$ are patterns. If prf_1 is not of the form described above, the parameters f and g are retained in the proof. Provided that the heads of f and g are constants, we can also omit the parameters x and y, respectively. As an example, we consider the derivation of $f\ c\ s\ t\ (g\ c\ u) \equiv f\ d\ s\ t\ (g\ d\ u)$ from $\mathit{prf} : c \equiv d$, where f, g, c and d are constants. The uncompressed version of the proof of the above equation, which is generated by Isabelle's term rewriting module, looks as follows:

$$\begin{array}{l}
\mathit{comb} \cdot f\,c\,s\,t \cdot f\,d\,s\,t \cdot g\,c\,u \cdot g\,d\,u \bullet \\
\ (\mathit{comb} \cdot f\,c\,s \cdot f\,d\,s \cdot t \cdot t \bullet \\
\quad (\mathit{comb} \cdot f\,c \cdot f\,d \cdot s \cdot s \bullet \\
\qquad (\mathit{comb} \cdot f \cdot f \cdot c \cdot d \bullet (\mathit{refl} \cdot f) \bullet \mathit{prf}) \bullet \\
\qquad (\mathit{refl} \cdot s)) \bullet \\
\quad (\mathit{refl} \cdot t)) \bullet \\
\ (\mathit{comb} \cdot g\,c \cdot g\,d \cdot u \cdot u \bullet (\mathit{comb} \cdot g \cdot g \cdot c \cdot d \bullet (\mathit{refl} \cdot g) \bullet \mathit{prf}) \bullet \\
\quad (\mathit{refl} \cdot u))
\end{array}$$

Theorem name	Full proof		Current impl.		Static		Refined		Bidirectional	
	terms	size	size	ratio	size	ratio	size	ratio	size	ratio
IntArith.pos-zmult-eq-1-iff	49046	247726	5615	97.73	119232	51.87	46386	81.28	19892	91.97
IntDiv.pos-zdiv-mult-2	46985	245920	6218	97.47	125602	48.93	47319	80.76	18875	92.32
IntDiv.pos-zmod-mult-2	33013	175819	4457	97.47	89557	49.06	33708	80.83	13661	92.23
Presburger.int-ge-induct	29603	206508	4441	97.85	85474	58.61	29849	85.55	19000	90.80
List.nibble.split	27938	114541	1792	98.44	46086	59.76	19056	83.36	6824	94.04
IntDiv.self-quotient	27310	149712	3722	97.51	72859	51.33	27437	81.67	12256	91.81
IntArith.int-le-induct	25381	191215	4055	97.88	78631	58.88	26390	86.20	17933	90.62
IntArith.int-ge-induct	24962	187436	3962	97.89	76442	59.22	25854	86.21	17658	90.58
List.list-all2-append1	23900	116870	2821	97.59	53897	53.88	22532	80.72	8511	92.72
IntDiv.divAlg-correct	23886	117037	2892	97.53	54007	53.85	22355	80.90	8520	92.72
List.nibble.split-asm	23762	116263	1808	98.44	50540	56.53	17344	85.08	7263	93.75
List.nth-upt	21721	101355	2454	97.58	47273	53.36	21092	79.19	7014	93.08
IntDiv.zminus1-lemma	18485	99613	3082	96.91	45480	54.34	18343	81.59	7993	91.98
IntDiv.zadd1-lemma	17716	94929	3554	96.26	45242	52.34	18010	81.03	7745	91.84
IntDiv.quorem-neg	17329	91425	2678	97.07	43515	52.40	17365	81.01	7420	91.88
IntArith.nat-abs-mult-distrib	17253	122440	2291	98.13	49490	59.58	16836	86.25	10182	91.68
IntArith.triangle-ineq	14042	87446	1868	97.86	38984	55.42	13765	84.26	7539	91.38
IntArith.abs-mult	13490	73762	1754	97.62	34657	53.02	13720	81.40	5384	92.70
Presburger.decr-lemma	12994	99165	1901	98.08	44937	54.68	13644	86.24	8902	91.02
Presburger.cpmi-eq	12490	151005	1867	98.76	49780	67.03	13521	91.05	15436	89.78
Presburger.cppi-eq	12490	151005	1867	98.76	49780	67.03	13521	91.05	15436	89.78
NatBin.zpower-number-of-odd	11857	73052	1749	97.61	36196	50.45	13173	81.97	5949	91.86
IntDiv.dvd-int-iff	11612	52667	1316	97.50	24350	53.77	10199	80.63	4088	92.24
Presburger.incr-lemma	11452	85426	1731	97.97	36992	56.70	11573	86.45	7917	90.73
List.nth-list-update-neq	10440	48047	1246	97.41	22260	53.67	10003	79.18	3525	92.66
IntArith.abs-abs	10279	49475	1164	97.65	20419	58.73	7845	84.14	3993	91.93
IntDiv.zdiv-mono2-lemma	10223	54615	1186	97.83	26626	51.25	9851	81.96	4477	91.80
List.upt-rec	9766	45499	1048	97.70	19772	56.54	8379	81.58	3494	92.32
Presburger.nnf-sdj	9349	44312	3612	91.85	4056	90.85	2080	95.31	3694	91.66
IntDiv.zmult1-lemma	9187	54221	1960	96.39	25030	53.84	9547	82.39	4630	91.46

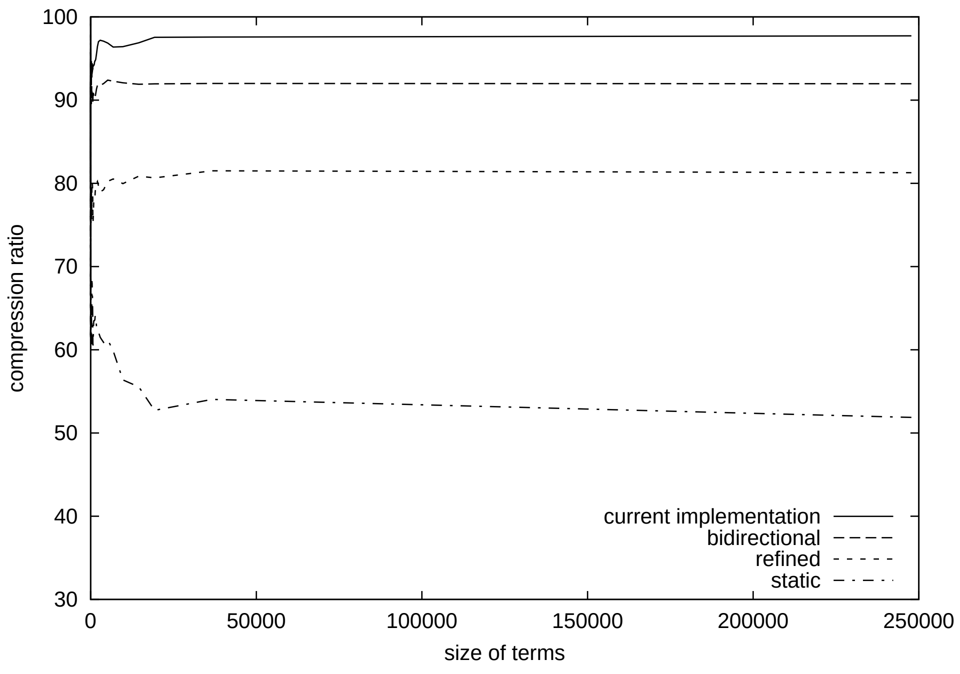


Figure 2.12: Performance of different compression algorithms

By applying the technique described above, we can compress this to the proof

$$
\begin{array}{l}
\mathit{comb} \cdot \_ \cdot \_ \cdot \_ \cdot \_ \bullet \\
\ (\mathit{comb} \cdot \_ \cdot \_ \cdot \_ \cdot \_ \bullet \\
\quad (\mathit{comb} \cdot \_ \cdot \_ \cdot \_ \cdot \_ \bullet \\
\qquad (\mathit{comb} \cdot f \cdot \_ \cdot \_ \cdot \_ \bullet (\mathit{refl} \cdot \_) \bullet \mathit{prf}) \bullet \\
\qquad (\mathit{refl} \cdot \_)) \bullet \\
\quad (\mathit{refl} \cdot \_)) \bullet \\
\ (\mathit{comb} \cdot \_ \cdot \_ \cdot \_ \cdot \_ \bullet (\mathit{comb} \cdot g \cdot \_ \cdot \_ \cdot \_ \bullet (\mathit{refl} \cdot \_) \bullet \mathit{prf}) \bullet \\
\quad (\mathit{refl} \cdot \_))
\end{array}
$$

By propagating the information in the proof downwards from the leaves to the root, we can infer that the proved equation has the form

$$f\ c\ ?x\ ?y\ (g\ c\ ?z) \equiv f\ d\ ?x\ ?y\ (g\ d\ ?z)$$

which is a pattern, where $?x$, $?y$ and $?z$ are unification variables introduced during reconstruction. Interestingly, this relatively simple optimization turned out to be the most effective compression strategy, resulting in a compression ratio of about 95%. This is due to the dominance of rewriting in most proofs. In the whole Isabelle/HOL library, the *comb* rule is used about 160000 times, reflexivity about 150000 times and transitivity about 60000 times. In contrast, ordinary natural deduction rules are used much less frequently. For example, the implication introduction rule is only used about 8000 times.

2.5 Related work

Dowek [34] presents a proof synthesis method for the systems of the λ-cube. This also includes an encoding of resolution in type theory similar to the one described in §2.3.1, which has found its way into the theorem prover Coq [12, §7.3] in the form of the `Intro` and `Apply` tactics.

Pfenning [94] describes several algorithms for unification and proof search in the Elf system, an implementation of the LF logical framework. The algorithm for the reconstruction of implicit arguments used in Elf, which is only sketched in [94, §5.5], seems to be quite comparable to ours (§2.4.1). Both in Elf and the more recent Twelf implementation [99], the user can mark certain outermost quantifiers in the type of a constant as *explicit*, while leaving others *implicit*. When applying a constant, only the arguments corresponding to explicit quantifiers need to be given, and the type checker automatically inserts placeholders for arguments corresponding to implicit quantifiers. For example, in the example signature defining first order logic, which is part of the Twelf distribution, the declaration for the existential introduction rule is

```
existsi : {T:i} nd (A T) -> nd (exists A).
```

where `{T:i}` denotes a *dependent product*, which plays the role of a universal quantifier, and `i` is the type of *individuals*. Here, only the witness `T` for the existential statement has to be given explicitly, while the formula `A` denoting the body of the existential quantifier is left implicit. As noted in §2.4.5, this annotation only guarantees successful reconstruction of a term, if the direction of dataflow is bottom-up. In contrast to Isabelle, Twelf currently does not determine automatically which arguments of a constant have to be made explicit in order to ensure successful reconstruction. As in Isabelle, the user may also freely use placeholders _ to be filled in by the reconstruction algorithm elsewhere in a term.

Similar but more restricted strategies for synthesizing arguments, which are usually referred to as *implicit syntax*, have also been implemented by Pollack [102] in the LEGO proof assistant based on the Extended Calculus of Constructions (ECC), and, albeit in a somewhat crude form, in Coq [12]. Pollack's calculus for *uniform argument synthesis* contains two kinds of abstractions $[x : A]M$ and $[x \mid A]M$, as well as two product types $\{x : A\}B$ and $\{x \mid A\}B$, corresponding to functions with explicit and implicit arguments, respectively. Pollack specifies a transformation for turning implicit terms into placeholder-free ones on a relative abstract level, but does not describe how to actually reconstruct omitted terms, e.g. via constraint collection and unification as described in §2.4.1. A similar *implicit* version of the Calculus of Constructions is also proposed by Miquel [70], although, as he admits, his system seems to be a poor candidate for being used in a proof assistant, since it is unclear under which conditions type checking is decidable.

Luther [64] develops compression and reconstruction algorithms for the ECC, which improve on the algorithms for argument synthesis currently implemented in LEGO or Coq. Similar to the bidirectional algorithm presented in §2.4.5, Luther's reconstruction and compression algorithms operate either in *checking* or *synthesis* mode, which corresponds to bottom-up and top-down dataflow, respectively.

Chapter 3

Proofs for equational logic

3.1 Introduction

Equational reasoning and rewriting play a central role in theorem proving. While there are specific systems tailored to support proofs in equational logic, such as ELAN [23], or in first order logic with equality, such as Otter [66] or Spass [118], it is also indispensable for a general purpose theorem prover like Isabelle or HOL to offer support for conducting proofs involving equational reasoning. One of the first implementations of rewriting in an interactive theorem prover was done by Paulson [89] in the Cambridge LCF system.

This section is concerned with a description of the term rewriting algorithm used in Isabelle. The previous implementation of rewriting, due to Tobias Nipkow, was part of Isabelle's trusted kernel. In contrast, the present implementation described here actually generates proofs and can therefore be implemented outside the kernel. Moreover, it improves on the old implementation by allowing contextual rewriting with unlimited mutual simplification of premises.

Term rewriting consists in replacing terms by *equal* terms using a set of *rewrite rules*. These are equations of the form $t \equiv u$, which are considered to be *directed*, i.e. they are applied in a left-to-right manner. Rewriting is justified by the rules of equational logic. The equational rules used in Isabelle/Pure are shown in Figure 3.1. We will sometimes refer to $\equiv$ as *meta-equality*, to distinguish it from other notions of equality defined in object logics. While the rules refl, sym and trans just state that $\equiv$ is an equivalence relation, the rules comb and abs, which state that $\equiv$ is a *congruence* with respect to function application and abstraction, are used to justify the application of rewriting steps in subterms. A specific property of simply-typed higher order logic as implemented in Isabelle/Pure is that the type *prop* of propositions is a type like any other. Consequently, $\equiv$ can also be used to express equality of propositions.

$$\frac{}{x \equiv x}\ \mathsf{refl} \qquad \frac{x \equiv y}{y \equiv x}\ \mathsf{sym} \qquad \frac{x \equiv y \quad y \equiv z}{x \equiv z}\ \mathsf{trans}$$

$$\frac{f \equiv g \quad x \equiv y}{f\ x \equiv g\ y}\ \mathsf{comb} \qquad \frac{\bigwedge x.\ f\ x \equiv g\ x}{(\lambda x.\ f\ x) \equiv (\lambda x.\ g\ x)}\ \mathsf{abs}$$

$$\frac{P \Longrightarrow Q \quad Q \Longrightarrow P}{P \equiv Q}\ \mathsf{eqI} \qquad \frac{P \equiv Q \quad P}{Q}\ \mathsf{eqE}$$

Figure 3.1: Equational rules for Isabelle/Pure

```
$\mathsf{rewc}\ \overline{r}\ t \;=\; \begin{cases} \lfloor\theta(eq)\rfloor & \text{if } eq \in \overline{r} \text{ and } t = \theta(\mathsf{lhs}\ eq) \\ \bot & \text{otherwise} \end{cases}$

$\mathsf{botc}\ \overline{r}\ t$ = case $\mathsf{subc}\ \overline{r}\ t$ of
            $\bot \Rightarrow$ (case $\mathsf{rewc}\ \overline{r}\ t$ of
                $\bot \Rightarrow \bot$
              | $\lfloor eq_2 \rfloor \Rightarrow \mathsf{trans}\ \lfloor eq_2 \rfloor\ (\mathsf{botc}\ \overline{r}\ (\mathsf{rhs}\ eq_2))$)
          | $\lfloor eq_1 \rfloor \Rightarrow$ (case $\mathsf{rewc}\ \overline{r}\ (\mathsf{rhs}\ eq_1)$ of
                $\bot \Rightarrow \lfloor eq_1 \rfloor$
              | $\lfloor eq_2 \rfloor \Rightarrow \mathsf{trans}\ (\mathsf{trans}\ \lfloor eq_1 \rfloor\ \lfloor eq_2 \rfloor)\ (\mathsf{botc}\ \overline{r}\ (\mathsf{rhs}\ eq_2))$)

$\mathsf{subc}\ \overline{r}\ (\lambda x.\ t)$       = (case $\mathsf{botc}\ \overline{r}\ t$ of
                              $\bot \Rightarrow \bot$
                            | $\lfloor eq \rfloor \Rightarrow \lfloor \mathsf{abs}\ x\ eq \rfloor$)
$\mathsf{subc}\ \overline{r}\ ((\lambda x.\ t)\ u)$   = (case $\mathsf{subc}\ \overline{r}\ (\mathsf{rhs}\ eq_1)$ of
                              $\bot \Rightarrow \lfloor eq_1 \rfloor$
                            | $\lfloor eq_2 \rfloor \Rightarrow \mathsf{trans}\ \lfloor eq_1 \rfloor\ \lfloor eq_2 \rfloor$)
                          where $eq_1 = \mathsf{beta}\ ((\lambda x.\ t)\ u)$
$\mathsf{subc}\ \overline{r}\ (\overline{P} \Longrightarrow Q)$   = $\mathsf{impc}\ \overline{r}\ []\ \overline{P}\ Q\ []$
$\mathsf{subc}\ \overline{r}\ (t\ u)$         = (case $\mathsf{botc}\ \overline{r}\ t$ of
                              $\bot \Rightarrow$ (case $\mathsf{botc}\ \overline{r}\ u$ of
                                  $\bot \Rightarrow \bot$
                                | $\lfloor eq_2 \rfloor \Rightarrow \lfloor \mathsf{comb}\ (\mathsf{refl}\ t)\ eq_2 \rfloor$)
                            | $\lfloor eq_1 \rfloor \Rightarrow$ (case $\mathsf{botc}\ \overline{r}\ u$ of
                                  $\bot \Rightarrow \lfloor \mathsf{comb}\ eq_1\ (\mathsf{refl}\ u) \rfloor$
                                | $\lfloor eq_2 \rfloor \Rightarrow \lfloor \mathsf{comb}\ eq_1\ eq_2 \rfloor$))
```

Figure 3.2: Basic bottom-up rewriting algorithm

The fact that $P \equiv Q$ for P and Q of type *prop* just means *"if and only if"* is reflected by the rules eqI and eqE. This is in contrast to the usual definition of equality which one can find in type theories such as the Calculus of Constructions, where $\equiv : \alpha \to \alpha \to \mathsf{Prop}$ and $\alpha : \mathsf{Prop}$, where Prop is the universe of types (or propositions). In this case, α cannot be instantiated with Prop, since we do not have Prop : Prop but Prop : Type. This treatment of equality of propositions in Isabelle/Pure is not completely unproblematic, as we will show later.

3.2 Contextual rewriting

Isabelle's rewriting engine is based on so-called *conversions*, which were first introduced by Paulson in the LCF system [89]. A conversion is a function of type `term` $\Rightarrow$ `thm` which, given a term t, returns a theorem $t \equiv u$. The rewriting strategy used in Isabelle is bottom-up. This corresponds to the call-by-value evaluation scheme found e.g. in the functional programming language ML. In the presence of rewrite rules with several occurrences of the same variable on the right-hand side, this strategy avoids having to rewrite identical terms several times, but may also lead to unnecessary rewriting steps when a variable occurring on the left-hand side of a rewrite rule does not occur on the right-hand side. The basic rewriting algorithm is shown in Figure 3.2. It consists of three conversions rewc, botc and subc. All of these conversions have a set of rewrite rules $\overline{r}$ as an argument. In order to signal a failed attempt to rewrite a term, they actually return elements of an *option* datatype with constructors $\bot$ and $\lfloor\_\rfloor$. To reduce

$$\frac{A \Longrightarrow B \equiv B'}{(A \Longrightarrow B) \equiv (A \Longrightarrow B')}\ \mathsf{imp\_cong}_1 \qquad \frac{B \Longrightarrow (A \Longrightarrow C) \equiv (A' \Longrightarrow C)}{(A \Longrightarrow B \Longrightarrow C) \equiv (A' \Longrightarrow B \Longrightarrow C)}\ \mathsf{imp\_cong}_2$$

$$\frac{A \equiv A'}{(A \Longrightarrow B) \equiv (A' \Longrightarrow B)}\ \mathsf{imp\_cong}_3$$

Figure 3.3: Congruence rules for $\Longrightarrow$

the number of case distinctions in the above presentation of the algorithm, we extend trans to operate on the option datatype, i.e.

$$\begin{array}{llll} \mathsf{trans} & \bot & \bot & = \bot \\ \mathsf{trans} & \lfloor eq_1 \rfloor & \bot & = \lfloor eq_1 \rfloor \\ \mathsf{trans} & \bot & \lfloor eq_2 \rfloor & = \lfloor eq_2 \rfloor \end{array}$$

Moreover, we define a lifting operator $(\_)_\bot$ on functions as follows:

$$\begin{array}{lll} (f)_\bot\ \bot & = & \bot \\ (f)_\bot\ \lfloor x \rfloor & = & \lfloor f\ x \rfloor \end{array}$$

We also use the functions lhs and rhs for mapping a theorem of the form $t \equiv u$ to the terms t and u, respectively. Conversion rewc tries to apply a rewrite rule eq from $\overline{r}$ at the top level of term t. This involves finding a most general matcher θ of t and the left-hand side of eq, which is computed using a variant of the unification algorithm for *higher order patterns* proposed by Miller [69, 79]. The main work is done by the mutually recursive conversions botc and subc, where the former serves as the entry point. By calling subc, conversion botc first descends recursively into the term t to be rewritten and then tries to apply a rewrite rule at the top level using rewc. If a rewrite rule was applicable, the whole process is started all over again with the new term. Conversion subc decomposes the term to be rewritten and recursively applies botc to its subterms. The equation for rewriting the input term is then obtained from the equations for the subterms by applying the congruence rules abs or comb from Figure 3.1. Moreover, subc also applies β-reduction steps if possible. Terms of the form $P_1 \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow Q$ receive a special treatment. When rewriting a formula of this kind, there are essentially two main cases to consider:

- The premises $P_1, \ldots, P_n$ may be used to rewrite each other, i.e. any premise P_j may be used in the derivation of an equation $P_i \equiv P_i'$ where $i \neq j$.
- Premises may be used in rewriting the conclusion, i.e. in the derivation of an equation $(P_i \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow Q) \equiv R$ we may use any P_j with $j < i$.

This kind of *contextual rewriting* is justified by specific congruence rules for $\Longrightarrow$, which are shown in Figure 3.3. They are easily derived from the basic rules for equality shown in Figure 3.1. Rewriting of premises in general is justified by rule imp_cong$_3$. Using a premise P_j in rewriting a premise P_i can be justified by the congruence rule imp_cong$_1$ if $j < i$, i.e. the premise to be rewritten is to the right of the premise used in deriving the equation $P_i \equiv P_i'$, whereas imp_cong$_2$ can be used as a justification if $i < j$, i.e. the premise to be rewritten is to the left of the premise used. Rule imp_cong$_1$ also serves as a justification for using premises in rewriting the conclusion. Rewriting of implications is performed by the conversion impc, which is shown in Figure 3.4. To write down the algorithm, we introduce a little more notation. We use [] to denote the empty list. The notation (_ @ _) is used both for appending two lists, as well

```
$\mathsf{disch}\ P\ eq \;=\; \mathsf{imp\_cong_1}\ (\mathsf{implies\_intr}\ P\ eq)$

$\mathsf{disch_r}\ P\ eq \;=\; \mathsf{imp\_cong_2}\ (\mathsf{implies\_intr}\ P\ eq)$

$\mathsf{rebuild}\ \overline{r}\ []\ Q\ eq = eq$

$\mathsf{rebuild}\ \overline{r}\ (\overline{P}.P)\ Q\ eq\ =$
    let $R = \begin{cases} P \Longrightarrow Q & \text{if } eq = \bot \\ P \Longrightarrow \mathsf{rhs}\ eq' & \text{if } eq = \lfloor eq' \rfloor \end{cases}$
    in case $\mathsf{rewc}\ (\overline{r} \cup \mathsf{rews\_of}\ \overline{P})\ R$ of
        $\bot \Rightarrow \mathsf{rebuild}\ \overline{r}\ \overline{P}\ R\ ((\mathsf{disch}\ P)_\bot\ eq)$
      $|$ $\lfloor eq' \rfloor \Rightarrow$
        let $\overline{P'} \Longrightarrow Q' = \mathsf{rhs}\ eq'$
        in $\mathsf{trans}\ (\mathsf{fold}\ \mathsf{disch}\ \overline{P}\ (\mathsf{trans}\ ((\mathsf{disch}\ P)_\bot\ eq)\ \lfloor eq' \rfloor))\ (\mathsf{impc}\ \overline{r}\ []\ (\overline{P}.\overline{P'})\ Q'\ [])$

$\mathsf{impc}\ \overline{r}\ \overline{P}\ []\ Q\ \overline{eq} =$ (case $\mathsf{fold}\ (\lambda(eq_1,\ P)\ eq_2.\ \mathsf{trans}\ eq_1\ ((\mathsf{disch}\ P)_\bot\ eq_2))\ \bot\ (\mathsf{zip}\ \overline{eq}\ \overline{P})$ of
        $\bot \Rightarrow \mathsf{rebuild}\ \overline{r}\ \overline{P}\ Q\ (\mathsf{botc}\ (\overline{r} \cup \mathsf{rews\_of}\ \overline{P})\ Q)$
      $|$ $\lfloor eq \rfloor \Rightarrow \mathsf{trans}\ \lfloor eq \rfloor\ (\mathsf{impc}\ \overline{r}\ []\ \overline{P}\ Q\ []))$

$\mathsf{impc}\ \overline{r}\ \overline{P}\ (P.\overline{P_r})\ Q\ \overline{eq} =$ (case $\mathsf{botc}\ (\overline{r} \cup \mathsf{rews\_of}\ (\overline{P} \cup \overline{P_r}))\ P$ of
        $\bot \Rightarrow \mathsf{impc}\ \overline{r}\ (\overline{P}.P)\ \overline{P_r}\ (\overline{eq}.\bot)$
      $|$ $\lfloor eq \rfloor \Rightarrow \mathsf{impc}\ \overline{r}\ (\overline{P}.\mathsf{rhs}\ eq)\ \overline{P_r}\ \left(\overline{eq}.\ \lfloor \mathsf{fold}\ \mathsf{disch_r}\ \overline{P_r}\ (\mathsf{imp\_cong_3}\ Q\ eq) \rfloor\right))$
```

Figure 3.4: Contextual rewriting

as for inserting a single element at the head or at the end of a list. Since we use e.g. $\overline{P}$ to denote a list and P to denote a list element, it is clear what version of (_._) we are referring to. We also introduce the operations

$$
\begin{array}{lcl}
\mathsf{fold}\ f\ []\ b & = & b \\
\mathsf{fold}\ f\ (a.\overline{a})\ b & = & f\ a\ (\mathsf{fold}\ f\ \overline{a}\ b) \\
\\
\mathsf{zip}\ []\ [] & = & [] \\
\mathsf{zip}\ (a.\overline{a})\ (b.\overline{b}) & = & (a,b).\mathsf{zip}\ \overline{a}\ \overline{b}
\end{array}
$$

which are well-known in functional programming. Rewriting of premises proceeds from left to right. To accomplish this, impc takes two lists of premises as an argument, the first of which is initially empty. The first list contains all premises $P_1 \ldots P_{i-1}$ which are to the left of the premise which is currently being rewritten, whereas the second list $P_i \ldots P_n$ contains the current premise as well as all premises to the right. In order to use a premise in rewriting another premise, we first have to extract rewrite rules from it. This is done by function rews_of. For example, the set of rewrite rules extracted from a premise of the form $A \wedge B$ is the union of the rewrite rules extracted from A and B. Moreover, a premise of the form $\neg A$ could be turned into the rewrite rule $A \equiv \mathit{False}$, whereas a non-negated premise P, which is not a conjunction, could be turned into the rewrite rule $P \equiv \mathit{True}$. In each step, impc tries to rewrite the first premise P_i from the second list and then appends a possibly rewritten version to the end of the first list. Each step produces an equation of the form $P_i \equiv P_i'$, whose proof may depend on assumptions $P_1' \ldots P_{i-1}'$ and $P_{i+1} \ldots P_n$. Assumptions $P_{i+1} \ldots P_n$, which may get rewritten in subsequent steps, are discharged immediately using $\mathsf{disch_r}$. This yields an equation eq_i of the form

$$(P_i \Longrightarrow P_{i+1} \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow Q) \equiv (P_i' \Longrightarrow P_{i+1} \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow Q)$$

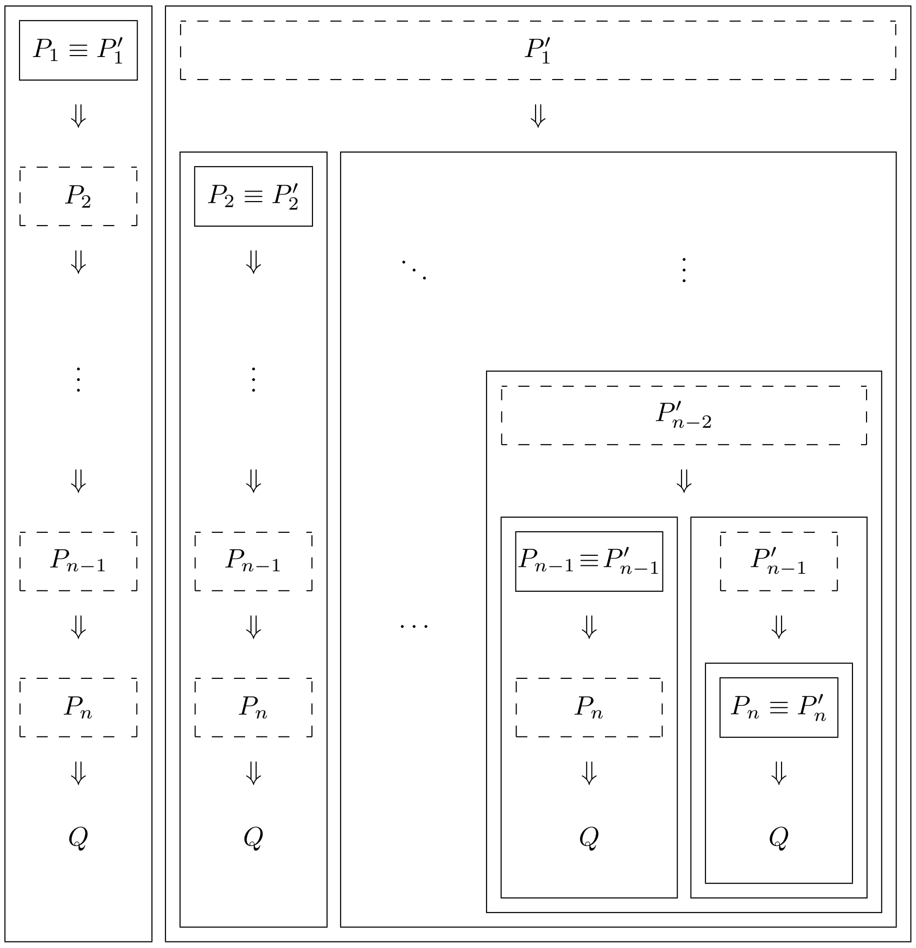


Figure 3.5: Mutual simplification of premises

The discharging of other assumptions is postponed until the end. The equations eq_i are accumulated in the list $\overline{eq}$. When all premises have been processed, the accumulated equations are put together using a kind of "staircase" technique illustrated in Figure 3.5. More precisely, we apply a sequence of discharging and transitivity operations, using the last equation eq_n as a starting point. By combining equations $eq_{i+1} \ldots eq_n$, we obtain the equation

$$(P_{i+1} \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow Q) \equiv (P'_{i+1} \Longrightarrow \cdots \Longrightarrow P'_n \Longrightarrow Q)$$

By discharging assumption P'_i, we get the equation

$$(P'_i \Longrightarrow P_{i+1} \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow Q) \equiv (P'_i \Longrightarrow P'_{i+1} \Longrightarrow \cdots \Longrightarrow P'_n \Longrightarrow Q)$$

Using transitivity together with eq_i, we can finally turn this into the equation

$$(P_i \Longrightarrow P_{i+1} \Longrightarrow \cdots \Longrightarrow P_n \Longrightarrow Q) \equiv (P'_i \Longrightarrow P'_{i+1} \Longrightarrow \cdots \Longrightarrow P'_n \Longrightarrow Q)$$

Note that by using this technique, we avoid having to discharge the hypotheses $P_1 \ldots P_{n-1}$ several times, i.e. for each of the equations $eq_2 \ldots eq_n$, which improves the efficiency of the algorithm. If some of the premises have been rewritten successfully, the list of premises is processed once more, since the rewrite rules extracted from the rewritten premises can give rise to new redexes. This is done by the recursive call in the first clause defining impc. Actually, we have to re-inspect at least all of the premises which are to the left of the rightmost premise just simplified. After rewriting the premises, the conclusion Q is rewritten, which yields an equation of the form $Q \equiv Q'$ whose derivation may depend on the assumptions $P'_1 \ldots P'_n$. Again, these assumptions need to be discharged. This is taken care of by function rebuild. After discharging an assumption P'_i, we also check whether a rewrite rule of the form $(\cdots \Longrightarrow \cdots) \equiv R$, whose left-hand side is an implication, such as $(\mathit{True} \Longrightarrow P) \equiv P$ is applicable to the term $P'_i \Longrightarrow \cdots \Longrightarrow P'_n \Longrightarrow Q'$. If this is the case, we obtain the equation

$$(P'_i \Longrightarrow \cdots \Longrightarrow P'_n \Longrightarrow Q) \equiv R$$

by transitivity, and hence

$$(P'_1 \Longrightarrow \cdots \Longrightarrow P'_n \Longrightarrow Q) \equiv (P'_1 \Longrightarrow \cdots \Longrightarrow P'_{i-1} \Longrightarrow R)$$

The application of a rule of the form $(\cdots \Longrightarrow \cdots) \equiv R$ may affect the premises and can, similar to the application of an equation $P_i \equiv P'_i$, give rise to new redexes due to rewrite rules extracted from the modified premises. If such a rule was applied, we therefore have to perform another iteration of the rewriting algorithm by calling function impc.

3.3 Transforming equational proofs

3.3.1 Rewriting on propositions

It has already been mentioned that $\equiv$ may also be used to express equality between propositions. While this seems to make proofs simpler at first sight, since we can use the equality rules uniformly on any type, it can be quite problematic from a proof-theoretic point of view. For example, we can construct a proof of the form

$$\begin{array}{l}\mathsf{eqE} \cdot (A \Longrightarrow B) \cdot (A' \Longrightarrow B') \bullet (\mathsf{comb} \cdot (\Longrightarrow A) \cdot (\Longrightarrow A') \cdot B \cdot B' \bullet \\ \quad (\mathsf{comb} \cdot \Longrightarrow \cdot \Longrightarrow \cdot A \cdot A' \bullet (\mathsf{refl} \cdot \Longrightarrow) \bullet \mathit{prf_1}) \bullet \mathit{prf_2}) \bullet \mathit{prf_3} \bullet \mathit{prf_4}\end{array}$$

where $\mathit{prf_1} : A \equiv A'$, $\mathit{prf_2} : B \equiv B'$, $\mathit{prf_3} : A \Longrightarrow B$ and $\mathit{prf_4} : A'$. Now the problem with the above proof is that it contains a potential redex, which is hidden by the comb rule. We can exhibit this redex by replacing the subproof containing comb by a more informative derivation using eqI and eqE:

$$\begin{array}{l}\mathsf{eqE} \cdot (A \Longrightarrow B) \cdot (A' \Longrightarrow B') \bullet \\ \quad (\mathsf{eqI} \cdot (A \Longrightarrow B) \cdot (A' \Longrightarrow B') \bullet \\ \qquad (\boldsymbol{\lambda} H_1 : (A \Longrightarrow B)\ H_2 : A'.\ \mathsf{eqE} \cdot B \cdot B' \bullet \mathit{prf_2} \bullet \\ \qquad\quad (H_1\ (\mathsf{eqE} \cdot A' \cdot A \bullet (\mathsf{sym} \cdot A \cdot A' \bullet \mathit{prf_1}) \bullet H_2))) \bullet \\ \qquad (\boldsymbol{\lambda} H_1 : (A' \Longrightarrow B')\ H_2 : A.\ \mathsf{eqE} \cdot B' \cdot B \bullet (\mathsf{sym} \cdot B \cdot B' \bullet \mathit{prf_2}) \bullet \\ \qquad\quad (H_1\ (\mathsf{eqE} \cdot A \cdot A' \bullet \mathit{prf_1} \bullet H_2)))) \bullet \mathit{prf_3} \bullet \mathit{prf_4}\end{array}$$

Now consider the following reduction rules for proofs involving eqI and eqE:

$$\mathsf{eqE} \cdot \_ \cdot \_ \bullet (\mathsf{eqI} \cdot \_ \cdot \_ \bullet \mathit{prf_1} \bullet \mathit{prf_2}) \longmapsto \mathit{prf_1}$$

$$\mathsf{eqE} \cdot \_ \cdot \_ \bullet (\mathsf{sym} \cdot \_ \cdot \_ \bullet (\mathsf{eqI} \cdot \_ \cdot \_ \bullet \mathit{prf_1} \bullet \mathit{prf_2})) \longmapsto \mathit{prf_2}$$

Using the first of these rules, the above proof finally reduces to

$$\mathsf{eqE} \cdot B \cdot B' \bullet \mathit{prf_2} \bullet (\mathit{prf_3} \bullet (\mathsf{eqE} \cdot A' \cdot A \bullet (\mathsf{sym} \cdot A \cdot A' \bullet \mathit{prf_1}) \bullet \mathit{prf_4}))$$

Since the intermediate step duplicates *prf₁* and *prf₂*, which can lead to an exponential blowup of the proof size, it is more advantageous to combine the two proof rewriting steps in a single rule, i.e.

$$\begin{array}{l}\mathsf{eqE} \cdot \_ \cdot \_ \bullet (\mathsf{comb} \cdot (\Longrightarrow A) \cdot (\Longrightarrow A') \cdot B \cdot B' \bullet \\ \quad (\mathsf{comb} \cdot \Longrightarrow \cdot \Longrightarrow \cdot A \cdot A' \bullet (\mathsf{refl} \cdot \Longrightarrow) \bullet \mathit{prf_1}) \bullet \mathit{prf_2}) \longmapsto \\ (\boldsymbol{\lambda} H_1 : (A \Longrightarrow B)\ H_2 : A'.\ \mathsf{eqE} \cdot B \cdot B' \bullet \mathit{prf_2} \bullet \\ \quad (H_1 \bullet (\mathsf{eqE} \cdot A' \cdot A \bullet (\mathsf{sym} \cdot A \cdot A' \bullet \mathit{prf_1}) \bullet H_2)))\end{array}$$

$$\begin{array}{l}\mathsf{eqE} \cdot \_ \cdot \_ \bullet (\mathsf{sym} \cdot \_ \cdot \_ \bullet (\mathsf{comb} \cdot (\Longrightarrow A) \cdot (\Longrightarrow A') \cdot B \cdot B' \bullet \\ \quad (\mathsf{comb} \cdot \Longrightarrow \cdot \Longrightarrow \cdot A \cdot A' \bullet (\mathsf{refl} \cdot \Longrightarrow) \bullet \mathit{prf_1}) \bullet \mathit{prf_2})) \longmapsto \\ (\boldsymbol{\lambda} H_1 : (A' \Longrightarrow B')\ H_2 : A.\ \mathsf{eqE} \cdot B' \cdot B \bullet (\mathsf{sym} \cdot B \cdot B' \bullet \mathit{prf_2}) \bullet \\ \quad (H_1 \bullet (\mathsf{eqE} \cdot A \cdot A' \bullet \mathit{prf_1} \bullet H_2)))\end{array}$$

We can perform similar transformations on proofs of equalities of the form $(\bigwedge x.\ P) \equiv (\bigwedge x.\ P')$. Note that in contrast to $\Longrightarrow$, the meta universal quantifier $\bigwedge$ is not even mentioned in the rewriting function presented in Figure 3.2, since rewriting under $\bigwedge$ can be justified using comb and abs. However, the usage of these rules is actually unnecessary in this case, since they can be eliminated by the following transformation:

$$\begin{array}{l}\mathsf{eqE} \cdot \_ \cdot \_ \bullet (\mathsf{comb} \cdot \bigwedge \cdot \bigwedge \cdot (\lambda x.\ P\ x) \cdot (\lambda x.\ P'\ x) \bullet (\mathsf{refl} \cdot \bigwedge) \bullet (\mathsf{abs} \cdot P \cdot P' \bullet \mathit{prf})) \longmapsto \\ (\boldsymbol{\lambda}(H : \bigwedge x.\ P\ x)\ x.\ \mathsf{eqE} \cdot P\ x \cdot P'\ x \bullet (\mathit{prf} \cdot x) \bullet (H \cdot x))\end{array}$$

$$\begin{array}{l}\mathsf{eqE} \cdot \_ \cdot \_ \bullet (\mathsf{sym} \cdot \_ \cdot \_ \bullet (\mathsf{comb} \cdot \bigwedge \cdot \bigwedge \cdot (\lambda x.\ P\ x) \cdot (\lambda x.\ P'\ x) \bullet (\mathsf{refl} \cdot \bigwedge) \bullet (\mathsf{abs} \cdot P \cdot P' \bullet \mathit{prf}))) \longmapsto \\ (\boldsymbol{\lambda}(H : \bigwedge x.\ P'\ x)\ x.\ \mathsf{eqE} \cdot P'\ x \cdot P\ x \bullet (\mathsf{sym} \cdot P\ x \cdot P'\ x \bullet (\mathit{prf} \cdot x)) \bullet (H \cdot x))\end{array}$$

We can also replace instances of trans and refl for type *prop* by alternative derivations:

$$\mathsf{eqE} \cdot A \cdot C \bullet (\mathsf{trans} \cdot A \cdot B \cdot C \bullet \mathit{prf_1} \bullet \mathit{prf_2}) \bullet \mathit{prf_3} \longmapsto \mathsf{eqE} \cdot B \cdot C \bullet \mathit{prf_2} \bullet (\mathsf{eqE} \cdot A \cdot B \bullet \mathit{prf_1} \bullet \mathit{prf_3})$$

$$\begin{array}{l}\mathsf{eqE} \cdot A \cdot C \bullet (\mathsf{sym} \cdot C \cdot A \bullet (\mathsf{trans} \cdot C \cdot B \cdot A \bullet \mathit{prf_1} \bullet \mathit{prf_2})) \bullet \mathit{prf_3} \longmapsto \\ \mathsf{eqE} \cdot B \cdot C \bullet (\mathsf{sym} \cdot C \cdot B \bullet \mathit{prf_1}) \bullet (\mathsf{eqE} \cdot A \cdot B \bullet (\mathsf{sym} \cdot B \cdot A \bullet \mathit{prf_2}) \bullet \mathit{prf_3})\end{array}$$

$$\mathsf{eqE} \cdot A \cdot A \bullet (\mathsf{refl} \cdot A) \bullet \mathit{prf} \longmapsto \mathit{prf}$$

$$\mathsf{eqE} \cdot A \cdot A \bullet (\mathsf{sym} \cdot A \cdot A \bullet (\mathsf{refl} \cdot A)) \bullet \mathit{prf} \longmapsto \mathit{prf}$$

3.3.2 Eliminating meta equality rules

The rewriting algorithm presented in the previous section only accepts meta-equalities $s \equiv t$ as rewrite rules. To allow object level equalities $s = t$ to be used as well, a rule of the form

$$\mathsf{eq\_reflection} :\ s = t \Longrightarrow s \equiv t$$

is needed, which states that equality on the object level implies equality on the meta level. Although the rule

$$\mathsf{meta\_eq\_to\_obj\_eq} :\ s \equiv t \Longrightarrow s = t$$

for the opposite direction is easily derived using the rules for meta-equality presented in Figure 3.1, together with the reflexivity rule for =, this is usually not the case for *eq_reflection*, except if the types for meta level and object level truth values coincide. For example, the object level substitution rule of the form $s = t \Longrightarrow P\ s \Longrightarrow P\ t$, which is used e.g. in HOL, requires P to have type $\alpha \Rightarrow \mathit{bool}$ and is therefore unsuitable to turn $s \equiv t$ into $t \equiv t$, since this would require P to be instantiated with $\lambda x.\ x \equiv t$, which is of type $\alpha \Rightarrow \mathit{prop}$. Hence, *eq_reflection* is usually assumed as an axiom.

To show that this axiom is actually admissible, we demonstrate that it can be eliminated from proofs by replacing meta level equality rules by object level equality rules. This can be done using the following set of rewrite rules for equational proofs:

$$(\mathsf{eqE} \cdot x_1 \cdot x_2 \bullet (\mathsf{comb} \cdot \mathsf{Tr} \cdot x_3 \cdot A \cdot B \bullet (\mathsf{refl} \cdot x_4) \bullet \mathit{prf_1}) \bullet \mathit{prf_2}) \longmapsto$$
$$(\mathsf{HOL.iffD}_1 \cdot A \cdot B \bullet (\mathsf{meta\_eq\_to\_obj\_eq} \cdot A \cdot B \bullet \mathit{prf_1}) \bullet \mathit{prf_2})$$

$$(\mathsf{eqE} \cdot x_1 \cdot x_2 \bullet (\mathsf{sym} \cdot x_3 \cdot x_4 \bullet (\mathsf{comb} \cdot \mathsf{Tr} \cdot x_5 \cdot A \cdot B \bullet (\mathsf{refl} \cdot x_6) \bullet \mathit{prf_1})) \bullet \mathit{prf_2}) \longmapsto$$
$$(\mathsf{HOL.iffD}_2 \cdot A \cdot B \bullet (\mathsf{meta\_eq\_to\_obj\_eq} \cdot A \cdot B \bullet \mathit{prf_1}) \bullet \mathit{prf_2})$$

$$(\mathsf{meta\_eq\_to\_obj\_eq} \cdot x_1 \cdot x_2 \bullet (\mathsf{comb} \cdot f \cdot g \cdot x \cdot y \bullet \mathit{prf_1} \bullet \mathit{prf_2})) \longmapsto$$
$$(\mathsf{HOL.cong} \cdot f \cdot g \cdot x \cdot y \bullet (\mathsf{meta\_eq\_to\_obj\_eq} \cdot f \cdot g \bullet \mathit{prf_1}) \bullet (\mathsf{meta\_eq\_to\_obj\_eq} \cdot x \cdot y \bullet \mathit{prf_2}))$$

$$(\mathsf{meta\_eq\_to\_obj\_eq} \cdot x_1 \cdot x_2 \bullet (\mathsf{trans} \cdot x \cdot y \cdot z \bullet \mathit{prf_1} \bullet \mathit{prf_2})) \longmapsto$$
$$(\mathsf{HOL.trans} \cdot x \cdot y \cdot z \bullet (\mathsf{meta\_eq\_to\_obj\_eq} \cdot x \cdot y \bullet \mathit{prf_1}) \bullet (\mathsf{meta\_eq\_to\_obj\_eq} \cdot y \cdot z \bullet \mathit{prf_2}))$$

$$(\mathsf{meta\_eq\_to\_obj\_eq} \cdot x \cdot x \bullet (\mathsf{refl} \cdot x)) \longmapsto (\mathsf{HOL.refl} \cdot x)$$

$$(\mathsf{meta\_eq\_to\_obj\_eq} \cdot x \cdot y \bullet (\mathsf{sym} \cdot x \cdot y \bullet \mathit{prf})) \longmapsto (\mathsf{HOL.sym} \cdot x \cdot y \bullet (\mathsf{meta\_eq\_to\_obj\_eq} \cdot x \cdot y \bullet \mathit{prf}))$$

$$(\mathsf{meta\_eq\_to\_obj\_eq} \cdot x_1 \cdot x_2 \bullet (\mathsf{abs} \cdot f \cdot g \bullet \mathit{prf})) \longmapsto$$
$$(\mathsf{HOL.ext} \cdot f \cdot g \bullet (\boldsymbol{\lambda} x.\ \mathsf{meta\_eq\_to\_obj\_eq} \cdot f\ x \cdot g\ x \bullet (\mathit{prf} \cdot x)))$$

$$(\mathsf{meta\_eq\_to\_obj\_eq} \cdot x \cdot y \bullet (\mathsf{eq\_reflection} \cdot x \cdot y \bullet \mathit{prf})) \longmapsto \mathit{prf}$$

The underlying idea is as follows: In order to rewrite an object logic goal of the form $\mathsf{Tr}\ A$, one can derive an equation of the form $\mathsf{Tr}\ A \equiv \mathsf{Tr}\ B$. Applying sym and eqE yields $\mathsf{Tr}\ B \Longrightarrow \mathsf{Tr}\ A$, which allows us to replace the original goal by the rewritten goal $\mathsf{Tr}\ B$. Assuming there are no other rules for deriving meta equalities except for eq_reflection and the rules from Figure 3.1, the only way to derive $\mathsf{Tr}\ A \equiv \mathsf{Tr}\ B$ is by finding a proof $\mathit{prf_1}$ of $A \equiv B$, from which we can obtain the aforementioned equation by applying the congruence rule comb. Since the equality of A and B may also be expressed on the object level, we may turn the meta level equality $A \equiv B$ into the object level equality $A = B$ using meta_eq_to_obj_eq and use iffD_2 instead of cong and eqE to obtain $\mathsf{Tr}\ A \equiv \mathsf{Tr}\ B$. This transformation is described by the second rule. The first rule describes the symmetric case of replacing a subgoal $\mathsf{Tr}\ B$ by $\mathsf{Tr}\ A$. The rule meta_eq_to_obj_eq is then simply pushed upwards in the proof tree, replacing the meta level equality rules comb, trans, refl, sym and abs by their object level conterparts HOL.cong, HOL.trans, HOL.refl, HOL.sym and HOL.ext, respectively. The last rule specifies what to do when meta_eq_to_obj_eq reaches a leaf of the equational proof consisting of an application of the rule eq_reflection to a proof *prf* of an object level equality. In this case, meta_eq_to_obj_eq absorbs eq_reflection, yielding just *prf*.

3.4 Related work

Boulton [24] describes a modular, conversion-based rewriting algorithm for the HOL theorem prover. His article mainly focuses on simplifying arithmetic expressions and therefore does not cover contextual rewriting or mutual simplification of premises. He also discusses several techniques for optimizing rewriting algorithms, such as avoiding processing of *unchanged* subterms.

For this problem, Boulton investigates several solutions, such as signaling unchanged terms via exceptions or by introducing a specific datatype similar to the one introduced in §3.2, with two constructors $\perp$ and $\lfloor \_ \rfloor$ corresponding to unchanged and changed terms, respectively. Apart from discussing the implementation of these approaches in detail, Boulton also gives a good survey of the history of rewriting algorithms in HOL and related systems. Boulton also points out that for the purpose of producing proof logs or viewing proofs as programs, not only the *execution time* of rewriting algorithms, but also the *size* of the produced equational proofs is of importance.

A description of rewriting algorithms from the perspective of constructive type theory can be found in Jackson's PhD thesis [54, §4], who presents an implementation of rewriting in the Nuprl proof development system.

The transformation of equational proofs by rewriting has also been studied by Nipkow [78, §3], although for slightly different purposes than described in §3.3. Thanks to the representation of proofs as terms, such transformations can now be performed inside Isabelle as well.

Chapter 4

Program extraction

4.1 Introduction

One of the most fascinating properties of constructive logic is that a proof of a specification contains an algorithm which, by construction, satisfies this specification. This idea forms the basis for *program extraction* mechanisms, which can be found in theorem provers such as Coq [12] or Nuprl [27]. To date, program extraction has mainly been restricted to theorem provers based on expressive dependent type theories such as the Calculus of Constructions [28]. A notable exception is the Minlog System by Schwichtenberg [15], which is based on minimal first order logic. Although Isabelle is based on simply-typed minimal higher order logic, which is purely constructive, little effort has been devoted to the issue of program extraction in this system so far.

The aim of this chapter is to demonstrate that Isabelle is indeed quite suitable as a basis for program extraction. Based on the encoding of proofs as λ-terms presented in §2.2, we describe a mechanism that turns an Isabelle proof into a functional program. Since Isabelle is a generic theorem prover, this mechanism will be generic, too. In order to instantiate it for a particular object logic, one has to assign programs to each of its primitive inference rules. By induction on the structure of proof terms, one can then build programs from more complex proofs making use of these inference rules. Since the essence of program extraction is to systematically produce programs that are *correct* by construction, we also describe a transformation that turns a proof into a correctness proof of the program extracted from it. The precise definition of what is meant by *correctness* will be given by a so-called *realizability interpretation*, that relates programs to logical formulae. The overall architecture of the program extraction framework is shown in Figure 4.1. It should be noted that the extracted program is actually available as a function in the object logic. Therefore, its proof of correctness can be checked inside Isabelle. The checking process turns the correctness proof into a genuine *theorem*, which may be used in other formalizations together with the extracted program. Finally, using Isabelle's code generator [20], the extracted function can be compiled into an efficiently executable ML program.

The rest of the chapter is structured as follows: In §4.2, the generic program extraction mechanism will be introduced, whereas §4.3 describes its adaption to Isabelle/HOL. This involves associating basic inference rules with suitable programs, as well as handling proofs involving more advanced constructs such as inductive datatypes and predicates. A suite of case studies is presented in the following Chapter 5.

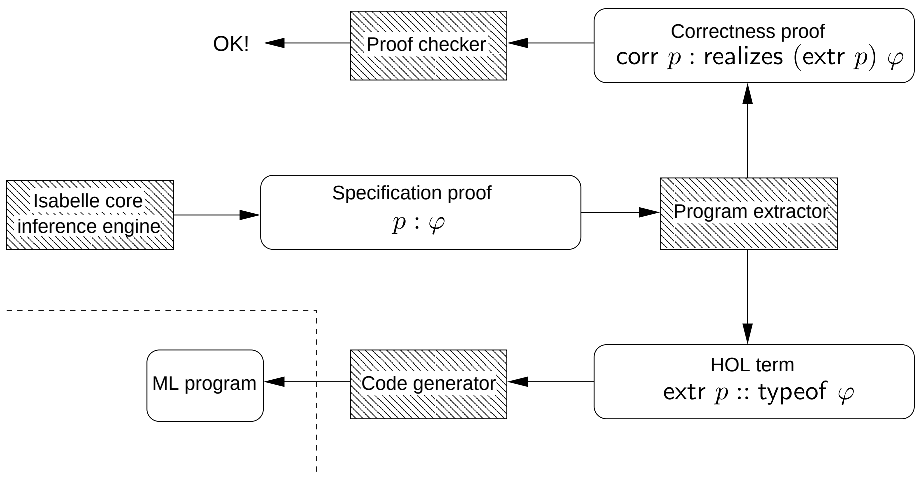


Figure 4.1: Architecture of the Isabelle program extraction framework

4.2 A generic framework for program extraction

We now come to the definition of the generic program extraction framework. As described in Figure 4.1, it consists of the following ingredients:

- A function typeof which maps a logical formula to the type of the term extracted from its proof
- The actual extraction function extr which extracts a term (i.e. a program) from a proof p with $\Gamma \vdash p : \varphi$, such that $\Gamma \vdash \mathsf{extr}\ \Gamma\ p :: \mathsf{typeof}\ \varphi$
- A function realizes which, given a term and a logical formula (the *specification*), returns a logical formula describing that the term in some sense satisfies (*"realizes"*) the specification
- A function corr which yields a proof that the program extracted from a proof p realizes the formula proved by p, i.e. $\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ p : \mathsf{realizes}\ (\mathsf{extr}\ \Gamma\ p)\ \varphi$

4.2.1 Extracting types

Terms of the λ-calculus contain typing information, and so does the program extracted from a proof. Therefore, the first step on the way from a proof to a program is the definition of a function computing this typing information. In order to determine the type of a program extracted from a proof $p : \varphi$, it suffices to analyze the proposition φ established by the proof. In the sequel, this analysis process will be referred to as *type extraction*. For example, intuition suggests that a program extracted from a proof of $P \Longrightarrow Q$ should be some function of type $\sigma \Rightarrow \tau$. An important observation in this context is that often only some parts of a formula correspond to actual computations, whereas other parts specify properties of the data manipulated by the algorithm underlying the proof, and therefore serve for justifying the correctness of the extracted program. For example, in a formula $x \neq 0 \Longrightarrow \cdots$, the premise $x \neq 0$ merely *verifies* that x has the right value. To capture such formulae without computational content, we introduce a dummy type Null, having the constant Null as its only inhabitant. One possibility of giving a type to the formula $x \neq 0 \Longrightarrow \cdots$ would be to simply assign it a function type with a dummy argument type, i.e. $\mathsf{Null} \Rightarrow \tau$. Unfortunately, such dummy arguments would lead to a considerable amount of garbage appearing in the extracted program. Even worse, when recursively extracting programs from lemmas appearing in a proof, one would be forced to extract useless dummy programs for all lemmas involved, regardless of their actual computational content. To remedy this, a more fine-grained definition of type extraction is needed. More precisely, the type corresponding to a formula will be specified by several rules, depending on the computational content of its subformulae. For example, if P is a formula without computational content, $P \Longrightarrow Q$ would simply correspond to the type τ instead of $\mathsf{Null} \Rightarrow \tau$. It is a central property of this optimized type extraction mechanism that neither the type nor the constant Null may actually occur in extracted programs.

Type extraction will be performed by the function typeof, which maps each term $P :: \tau$ to a type, where τ is of the form $\overline{\sigma} \Rightarrow \beta$ with $\beta \in \mathbb{P}$, and $\mathbb{P}$ denotes the set of *propositional types*, i.e. $\mathbb{P} = \{\mathsf{prop}, \mathsf{bool}, \ldots\}$. It is important that typeof also operates on function types, since it will also be used to extract types from term arguments of inference rules corresponding to predicate variables. This issue will be discussed in more detail later on. The function typeof is specified as a set of (conditional) rewrite rules. It can easily be adapted to specific object logics by adding new rules. The rules below specify the extracted type corresponding to formulae of Isabelle/Pure. They should be read like a functional program, i.e. earlier rules have precedence over rules appearing later.

$$\mathsf{typeof}\ P \equiv \mathsf{Null} \Longrightarrow \mathsf{typeof}\ Q \equiv \tau \Longrightarrow \mathsf{typeof}\ (P \Longrightarrow Q) \equiv \tau$$
$$\mathsf{typeof}\ Q \equiv \mathsf{Null} \Longrightarrow \mathsf{typeof}\ (P \Longrightarrow Q) \equiv \mathsf{Null}$$
$$\mathsf{typeof}\ P \equiv \sigma \Longrightarrow \mathsf{typeof}\ Q \equiv \tau \Longrightarrow \mathsf{typeof}\ (P \Longrightarrow Q) \equiv \sigma \Rightarrow \tau$$
$$(\bigwedge x.\ \mathsf{typeof}\ (P\ x) \equiv \mathsf{Null}) \Longrightarrow \mathsf{typeof}\ (\bigwedge x.\ P\ x) \equiv \mathsf{Null}$$
$$(\bigwedge x.\ \mathsf{typeof}\ (P\ x) \equiv \sigma) \Longrightarrow \mathsf{typeof}\ (\bigwedge x :: \alpha.\ P\ x) \equiv \alpha \Rightarrow \sigma$$
$$(\bigwedge x.\ \mathsf{typeof}\ (P\ x) \equiv \tau) \Longrightarrow \mathsf{typeof}\ P \equiv \tau$$

The rules describing the type extracted from $P \Longrightarrow Q$ are relatively straightforward. If P has no computational content, the type extracted from $P \Longrightarrow Q$ is just the type extracted from Q. If Q has no computational content, the whole formula has no computational content either. Otherwise, the type extracted from $P \Longrightarrow Q$ is a function type, where the argument type is the type extracted from P, while the result type is the type extracted from Q. The two rules for extracting the type of $(\bigwedge x.\ P\ x)$ are quite similar to the rules for implication. An important thing to note is the universal quantifier in the premise of the rules, which ensures that the type σ extracted from the body $P\ x$ of the quantifier does not depend on the value of the quantified variable x. The last rule describes the type extracted from a predicate, i.e. a term of type $\overline{\sigma} \Rightarrow \beta$, where $\overline{\sigma} \neq []$. This rule is useful for the extraction of types from the parameters of an inference rule (see the function TInst described below). As in the case of the universal quantifier, the type extracted from $P\ x$ must be the same for all values of x.

We also need to deal with predicate variables occurring in a formula. It depends on the formula a predicate variable is instantiated with, whether or not it contributes to the computational content of the formula it occurs in. If the variable is instantiated with a formula having computational content, we call the variable *computationally relevant*, otherwise *computationally irrelevant*. A computationally relevant predicate variable corresponds to a type variable in the type of the extracted program. During extraction, each computationally relevant predicate variable P is assigned a specific type variable α_P, i.e. $\mathsf{typeof}\ (P\ \overline{t}) \equiv \alpha_P$. In contrast,

$\mathsf{typeof}\ (Q\ \overline{t}) \equiv \mathsf{Null}$ for each computationally irrelevant variable Q. For a theorem with n predicate variables, there are 2^n possibilities for variables being computationally relevant or irrelevant. Thus, we may need to extract up to 2^n different programs from this theorem, depending on the context it is used in. For example, the program extracted from a proof of

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

will have type

$$(\alpha_P \Rightarrow \alpha_Q \Rightarrow \alpha_R) \Rightarrow (\alpha_P \Rightarrow \alpha_Q) \Rightarrow \alpha_P \Rightarrow \alpha_R$$

if P, Q and R are computationally relevant, whereas it will have type

$$(\alpha_Q \Rightarrow \alpha_R) \Rightarrow \alpha_Q \Rightarrow \alpha_R$$

if just Q and R are computationally relevant. Fortunately, only few of these variants are actually needed in practice, and our extraction mechanism can generate them on demand. Function RVars assigns to each theorem c with parameters $\overline{t}$ the set of its computationally relevant variables. Analogously, TInst yields a suitable type substitution for the type variables corresponding to computationally relevant predicate variables of c. Finally, we use PVars to denote the set of *all* predicate variables of a theorem c.

$$\begin{array}{lcl}
\mathsf{RVars}\ c\ \overline{t} & = & \{x_i \mid \Sigma(c) = (\bigwedge \overline{x} :: \overline{\tau}.\ \varphi),\ \tau_i = \overline{\sigma} \Rightarrow \beta,\ \beta \in \mathbb{P},\ \mathsf{typeof}\ t_i \neq \mathsf{Null}\} \\
\mathsf{TInst}\ c\ \overline{t} & = & \{\alpha_{x_i} \mapsto \tau \mid \Sigma(c) = (\bigwedge \overline{x} :: \overline{\tau}.\ \varphi),\ \tau_i = \overline{\sigma} \Rightarrow \beta,\ \beta \in \mathbb{P},\ \mathsf{typeof}\ t_i = \tau,\ \tau \neq \mathsf{Null}\} \\
\mathsf{PVars}\ c & = & \{x_i \mid \Sigma(c) = (\bigwedge \overline{x} :: \overline{\tau}.\ \varphi),\ \tau_i = \overline{\sigma} \Rightarrow \beta,\ \beta \in \mathbb{P}\}
\end{array}$$

From the implementation point of view, it is interesting to note that the rewrite rules for typeof given above can be formulated using Isabelle's term calculus introduced in §2.2.1. In order to explicitly encode *type* constraints on the level of *terms*, we use a technique originally introduced by Wenzel [119] for the purpose of formalizing *axiomatic type classes* in Isabelle. We introduce a new polymorphic type α itself together with a constant TYPE :: α itself. On top of this, we add a type Type together with a coercion function α itself $\Rightarrow$ Type. Then, typeof can be modelled as a function of type $\tau \Rightarrow$ Type, where τ is of the form $\overline{\sigma} \Rightarrow \beta$ with $\beta \in \mathbb{P}$. Using this formalism, equations of the form typeof $\varphi \equiv \tau$ occurring in the above presentation are actually encoded as typeof $\varphi \equiv$ Type (TYPE :: τ itself), where TYPE :: τ itself is usually abbreviated by TYPE(τ). It should be noted that the functions typeof and Type are not actually *defined* within Isabelle/Pure, since doing so would require a kind of *meta-logical framework* [99], but rather serve as syntax to formulate the rewrite rules above.

4.2.2 Extracting terms

We are now ready to give the definition of the extraction function extr. In addition to the actual proof, extr takes a context Γ as an argument, which associates term variables with types and proof variables with propositions. The extracted term is built up by recursion over the structure of a proof. The proof may refer to other theorems, for which we also need extracted programs. We therefore introduce a function $\mathcal{E}$ which maps a theorem name and a set of predicate variables to a term. We assume $\mathcal{E}$ to contain terms for both complex theorems, whose extracted term has already been computed by earlier invocations of extr, and primitive inference rules such as exI, for which a corresponding term has been specified by the author of the object logic. In the former case, the result of $\mathcal{E}$ will usually just be some constant referring to a more complex program, which helps to keep the extracted program more modular. As mentioned in §4.2.1, for theorems with predicate variables, the type of the corresponding program depends on the set of *relevant predicate variables*, which is passed as an additional argument to $\mathcal{E}$.

$$\begin{array}{lcl}
\mathsf{extr}\ \Gamma\ (c_{\{\overline{\alpha}\mapsto\overline{\tau}\}}\cdot\overline{t}) & = & \mathcal{E}(c, \mathsf{RVars}\ c\ \overline{t})(\{\overline{\alpha}\mapsto\overline{\tau}\}\cup\mathsf{TInst}\ c\ \overline{t})\ (\mathcal{E}_{\mathsf{args}}(\overline{t})) \\
\mathsf{extr}\ \Gamma\ h & = & \hat{h} \\
\mathsf{extr}\ \Gamma\ (\boldsymbol{\lambda}x :: \tau.\ p) & = & \lambda x :: \tau.\ \mathsf{extr}\ (\Gamma, x :: \tau)\ p \\
\mathsf{extr}\ \Gamma\ (\boldsymbol{\lambda}h : P.\ p) & = & \left\{\begin{array}{ll} \mathsf{extr}\ (\Gamma, h : P)\ p & \text{if } \tau = \mathsf{Null} \\ \boldsymbol{\lambda}\hat{h} :: \tau.\ \mathsf{extr}\ (\Gamma, h : P)\ p & \text{otherwise} \end{array}\right. \\
 & & \text{where } \mathsf{typeof}\ P = \tau \\
\mathsf{extr}\ \Gamma\ (p\cdot t) & = & (\mathsf{extr}\ \Gamma\ p)\ t \\
\mathsf{extr}\ \Gamma\ (p_1 \cdot p_2) & = & \left\{\begin{array}{ll} \mathsf{extr}\ \Gamma\ p_1 & \text{if } \tau = \mathsf{Null} \\ (\mathsf{extr}\ \Gamma\ p_1)\ (\mathsf{extr}\ \Gamma\ p_2) & \text{otherwise} \end{array}\right. \\
 & & \begin{array}{ll} \text{where} & \Gamma\vdash p_2 : P \\ & \mathsf{typeof}\ P = \tau \end{array}
\end{array}$$

where

$$\begin{array}{lcl}
\mathcal{E}_{\mathsf{args}}\ [] & = & [] \\
\mathcal{E}_{\mathsf{args}}\ (t_\tau, \overline{t}) & = & \left\{\begin{array}{ll} \mathcal{E}_{\mathsf{args}}(\overline{t}) & \text{if } \tau = \overline{\sigma}\Rightarrow\beta,\ \beta\in\mathbb{P} \\ t_\tau, \mathcal{E}_{\mathsf{args}}(\overline{t}) & \text{otherwise} \end{array}\right.
\end{array}$$

The first clause of extr deals with *proof constants*, i.e. references to axioms or theorems. For a theorem $\Sigma(c) = \bigwedge\overline{x}.\ \varphi$, the computational relevance of predicate variables in $\overline{x}$, and hence the computational content of the whole theorem, depends on the structure of the formulae the predicate variables are instantiated with. This instantiation can be read off from the context of c, i.e. from its argument list $\overline{t}$. Using RVars, we can infer the set of relevant predicate variables from the argument list $\overline{t}$, which is then used to select a suitable variant from the set of programs corresponding to c. The selected program contains type variables $\overline{\alpha_P}$ corresponding to predicate variables $\overline{P}\subseteq\overline{x}$, for which an instantiation is computed from $\overline{t}$ using TInst. Although terms of propositional type (i.e. predicates) in the argument list $\overline{t}$ have an influence on the *type* of the extracted program, they do not constitute programs themselves and therefore must not occur in the extracted program. Therefore, such terms are filtered out using the function $\mathcal{E}_{\mathsf{args}}$. The second clause of extr says that proof variables become term variables in the extracted program. To avoid clashes with already existing term variables, we map each proof variable h to a term variable $\hat{h}$ that does not occur in the original proof. Abstractions on the proof level, i.e. introduction of $\bigwedge$ and $\Longrightarrow$, are turned into abstractions on the program level. In the case of a proof of $P\Longrightarrow Q$, where P has no computational content, the extracted program is a degenerate "function" with no arguments. Analogously, applications on the proof level, i.e. elimination of $\bigwedge$ and $\Longrightarrow$, are turned into applications on the program level. In the case of an elimination of $P\Longrightarrow Q$, where P has no computational content, the function argument is omitted. Note that the clause for proofs of the form $(p\cdot t)$ does not apply to proofs p whose head is a constant, since these are handled by the first clause.

4.2.3 Correctness and realizability

It has already been mentioned in §4.1 that for each extracted program, one can obtain a correctness proof. For this correctness proof to make sense, we first have to make clear what is actually meant by correctness. The key for understanding the correctness of extracted programs is the notion of *realizability*. Realizability establishes a connection between a program and its specification. More precisely, we will specify a predicate realizes which relates terms (so-called *realizers*) with logical formulae. The notion of realizability was first introduced by Kleene [59] to study the semantics of intuitionistic logic. In his original formulation, realizers were *Gödel numbers*, which were somewhat hard to work with. To improve on this, Kreisel introduced so-called *modified realizability*, where realizers were actual terms of a kind of programming language, namely Gödel's *system T*. Our characterization of realizability, as well as the one which is described by Schwichtenberg [15], is inspired by Kreisel's modified realizability.

The following set of conditional rewrite rules characterizes realizability for formulae of the meta-logic Isabelle/Pure. As before, earlier rules have higher priority.

$$\mathsf{typeof}\ P \equiv \mathsf{Null} \Longrightarrow \mathsf{realizes}\ r\ (P \Longrightarrow Q) \equiv (\mathsf{realizes}\ \mathsf{Null}\ P \Longrightarrow \mathsf{realizes}\ r\ Q)$$

$$\begin{array}{l}\mathsf{typeof}\ P \equiv \sigma \Longrightarrow \mathsf{typeof}\ Q \equiv \mathsf{Null} \Longrightarrow \\ \quad \mathsf{realizes}\ r\ (P \Longrightarrow Q) \equiv (\bigwedge x :: \sigma.\ \mathsf{realizes}\ x\ P \Longrightarrow \mathsf{realizes}\ \mathsf{Null}\ Q)\end{array}$$

$$\mathsf{realizes}\ r\ (P \Longrightarrow Q) \equiv (\bigwedge x.\ \mathsf{realizes}\ x\ P \Longrightarrow \mathsf{realizes}\ (r\ x)\ Q)$$

$$(\bigwedge x.\ \mathsf{typeof}\ (P\ x) \equiv \mathsf{Null}) \Longrightarrow \mathsf{realizes}\ r\ (\bigwedge x.\ P\ x) \equiv (\bigwedge x.\ \mathsf{realizes}\ \mathsf{Null}\ (P\ x))$$

$$\mathsf{realizes}\ r\ (\bigwedge x.\ P\ x) \equiv (\bigwedge x.\ \mathsf{realizes}\ (r\ x)\ (P\ x))$$

For example, in the third clause defining realizability for $P \Longrightarrow Q$, P can be thought of as a specification of the *input* of program r, whereas Q specifies its *output*.

It is important to note that the above rules for realizes are still insufficient to cover cases where *predicate variables* occur in the formula to be realized. For example, how can we express that the induction principle for natural numbers

$$P\ 0 \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n)) \Longrightarrow P\ n$$

is realized by the program r? Since the above induction rule can be used in many different contexts, we do not know what realizes $r\ (P\ x)$ actually means, since we do not know in advance what formula P will get instantiated with. To solve this problem, we replace all computationally *relevant* predicate variables P with n arguments by a new predicate variable P^R with $n + 1$ arguments, where the additional argument is the realizing term:

$$\mathsf{realizes}\ r\ (P\ \bar{t}) \equiv P^R\ r\ \bar{t} \qquad \text{if } r \neq \mathsf{Null}$$

Later on, when an instantiation for P is known, we can substitute $\lambda r\ \overline{x}.\ \mathsf{realizes}\ r\ (P\ \overline{x})$ for P^R. Similarly, a computationally *irrelevant* predicate variable P is replaced by a new predicate variable P^R with the same number of arguments, i.e.

$$\mathsf{realizes}\ \mathsf{Null}\ (P\ \bar{t}) \equiv P^R\ \bar{t}$$

Thus, the fact that r realizes the induction principle on natural numbers can be expressed as follows:

$$\begin{array}{l}
\mathsf{realizes}\ r\ (P\ 0 \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow P\ (Suc\ x)) \Longrightarrow P\ n)\ \equiv \\
\bigwedge p_0.\ \mathsf{realizes}\ p_0\ (P\ 0) \Longrightarrow (\bigwedge p_S.\ \mathsf{realizes}\ p_S\ (\bigwedge x.\ P\ x \Longrightarrow P\ (Suc\ x))\ \Longrightarrow \\
\qquad \mathsf{realizes}\ (r\ p_0\ p_S)\ (P\ n))\ \equiv \\
\bigwedge p_0.\ \mathsf{realizes}\ p_0\ (P\ 0) \Longrightarrow (\bigwedge p_S.\ (\bigwedge x.\ \mathsf{realizes}\ (p_S\ x)\ (P\ x \Longrightarrow P\ (Suc\ x)))\ \Longrightarrow \\
\qquad \mathsf{realizes}\ (r\ p_0\ p_S)\ (P\ n))\ \equiv \\
\bigwedge p_0.\ \mathsf{realizes}\ p_0\ (P\ 0) \Longrightarrow (\bigwedge p_S.\ (\bigwedge x\ h.\ \mathsf{realizes}\ h\ (P\ x) \Longrightarrow \mathsf{realizes}\ (p_S\ x\ h)\ (P\ (Suc\ x)))\ \Longrightarrow \\
\qquad \mathsf{realizes}\ (r\ p_0\ p_S)\ (P\ n))\ \equiv \\
\bigwedge p_0.\ P^R\ p_0\ 0 \Longrightarrow (\bigwedge p_S.\ (\bigwedge x\ h.\ P^R\ h\ x \Longrightarrow P^R\ (p_S\ x\ h)\ (Suc\ x)) \Longrightarrow P^R\ (r\ p_0\ p_S)\ n)
\end{array}$$

We can now give a specification of function corr, which produces a correctness proof for the program computed by extr. It has a similar structure as function extr and again works by recursion on the proof. Since a proof may refer to other theorems, we also need a function $\mathcal{C}$ which yields correctness proofs for the programs extracted from these theorems. Its parameters are the same as those for function $\mathcal{E}$ described in §4.2.2.

$$\begin{array}{lcl}
\mathsf{corr}\ \Gamma\ (c_{\{\overline{\alpha} \mapsto \overline{\tau}\}} \cdot \overline{t}) & = & \mathcal{C}(c, \mathsf{RVars}\ c\ \overline{t})(\{\overline{\alpha} \mapsto \overline{\tau}\} \cup \mathsf{TInst}\ c\ \overline{t}) \cdot \mathcal{R}_{\mathsf{args}}(\overline{t}) \\
\mathsf{corr}\ \Gamma\ h & = & h \\
\mathsf{corr}\ \Gamma\ (\boldsymbol{\lambda} x :: \tau.\ p) & = & \boldsymbol{\lambda} x :: \tau.\ \mathsf{corr}\ (\Gamma, x :: \tau)\ p \\
\mathsf{corr}\ \Gamma\ (\boldsymbol{\lambda} h : P.\ p) & = & \left\{ \begin{array}{ll} \boldsymbol{\lambda} h : \mathsf{realizes}\ \mathsf{Null}\ P.\ \mathsf{corr}\ (\Gamma, h : P)\ p & \text{if } \tau = \mathsf{Null} \\ \boldsymbol{\lambda}(\hat{h} :: \tau)\ (h : \mathsf{realizes}\ \hat{h}\ P).\ \mathsf{corr}\ (\Gamma, h : P)\ p & \text{otherwise} \end{array} \right. \\
 & & \text{where } \mathsf{typeof}\ P = \tau \\
\mathsf{corr}\ \Gamma\ (p \cdot t) & = & (\mathsf{corr}\ \Gamma\ p) \cdot t \\
\mathsf{corr}\ \Gamma\ (p_1 \cdot p_2) & = & \left\{ \begin{array}{ll} \mathsf{corr}\ \Gamma\ p_1 \cdot \mathsf{corr}\ \Gamma\ p_2 & \text{if } \tau = \mathsf{Null} \\ \mathsf{corr}\ \Gamma\ p_1 \cdot \mathsf{extr}\ \Gamma\ p_2 \cdot \mathsf{corr}\ \Gamma\ p_2 & \text{otherwise} \end{array} \right. \\
 & & \begin{array}{ll} \text{where} & \Gamma \vdash p_2 : P \\ & \mathsf{typeof}\ P = \tau \end{array}
\end{array}$$

where the function

$$\begin{array}{lcl}
\mathcal{R}_{\mathsf{args}}\ [] & = & [] \\
\mathcal{R}_{\mathsf{args}}\ (t_\tau, \overline{t}) & = & \left\{ \begin{array}{ll} \lambda \overline{x} :: \overline{\sigma}.\ \mathsf{realizes}\ \mathsf{Null}\ (t\ \overline{x}),\ \mathcal{R}_{\mathsf{args}}(\overline{t}) & \text{if } \tau = \overline{\sigma} \Rightarrow \beta,\ \beta \in \mathbb{P},\ \mathsf{typeof}\ t = \mathsf{Null} \\ \lambda(r :: \varrho)\ (\overline{x} :: \overline{\sigma}).\ \mathsf{realizes}\ r\ (t\ \overline{x}),\ \mathcal{R}_{\mathsf{args}}(\overline{t}) & \text{if } \tau = \overline{\sigma} \Rightarrow \beta,\ \beta \in \mathbb{P},\ \mathsf{typeof}\ t = \varrho \neq \mathsf{Null} \\ t,\ \mathcal{R}_{\mathsf{args}}(\overline{t}) & \text{otherwise} \end{array} \right.
\end{array}$$

is used to compute an instantiation for the parameters of the correctness theorems for programs corresponding to proof constants. The main correctness property relating functions extr and corr can now be stated as follows:

Theorem 4.1 (Correctness of program extraction) Let

$$\vdash \mathcal{C}(c,\ V) : \mathsf{realizes}\ (\mathcal{E}(c, V))\ (\Sigma(c))$$

for all c and $V \subseteq \mathsf{PVars}(c)$. Then

$$\begin{array}{ll}
\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ q : \mathsf{realizes}\ \mathsf{Null}\ \varphi & \text{if } \mathsf{typeof}\ \varphi = \mathsf{Null} \\
\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ q : \mathsf{realizes}\ (\mathsf{extr}\ \Gamma\ q)\ \varphi & \text{otherwise}
\end{array}$$

for all Γ, q and φ with $\Gamma \vdash q : \varphi$, where

$$\begin{array}{lcll}
\mathcal{R}\ [] & = & [] & \\
\mathcal{R}\ (x :: \tau, \Gamma) & = & (x :: \tau, \mathcal{R}(\Gamma)) & \\
\mathcal{R}\ (h : P, \Gamma) & = & \left\{ \begin{array}{l} (h : \mathsf{realizes\ Null}\ P, \mathcal{R}(\Gamma)) \\ (\hat{h} :: \tau, h : \mathsf{realizes}\ \hat{h}\ P, \mathcal{R}(\Gamma)) \end{array} \right. & \begin{array}{l} \text{if } \tau = \mathsf{Null} \\ \text{otherwise} \end{array} \\
 & & \text{where } \mathsf{typeof}\ P = \tau &
\end{array}$$

Function $\mathcal{R}$ is used to express that, when producing a correctness proof for q, one may already assume to have suitable realizers and correctness proofs for each assumption in Γ. Since extr and corr depend on context information $\bar{t}$ for theorems, we require that each occurrence of a theorem (or proof constant) in q is *fully applied*, i.e. each theorem has as many term arguments as it has outermost $\bigwedge$-quantifiers. In the proof of the correctness theorem, we need the following *substitution property* for $\mathsf{realizes}$:

- $(\mathsf{realizes}\ r\ \varphi)\{x \mapsto t_\tau\} = \mathsf{realizes}\ (r\{x \mapsto t_\tau\})\ (\varphi\{x \mapsto t_\tau\})$, if $\tau \neq \overline{\sigma} \Rightarrow \beta$, where $\beta \in \mathbb{P}$
- $(\mathsf{realizes}\ r\ \varphi)\{P^R \mapsto \lambda\overline{x} :: \overline{\sigma}.\ \mathsf{realizes\ Null}\ (\psi\ \overline{x})\} = \mathsf{realizes}\ r\ (\varphi\{P \mapsto \psi\})$, if $\mathsf{typeof}\ \varphi = \mathsf{Null}$
- $(\mathsf{realizes}\ r\ \varphi)\{P^R \mapsto \lambda(r :: \varrho)\ (\overline{x} :: \overline{\sigma}).\ \mathsf{realizes}\ r\ (\psi\ \overline{x})\} = \mathsf{realizes}\ r\ (\varphi\{P \mapsto \psi\})$, if $\mathsf{typeof}\ \varphi = \varrho \neq \mathsf{Null}$

The proof of the correctness theorem, of which we only show some particularly interesting cases, is by induction on the structure of the proof q.

Case $q = h$ Since $h : \varphi \in \Gamma$, we have $\mathcal{R}(\Gamma) \vdash h : \mathsf{realizes\ Null}\ \varphi$ or $\mathcal{R}(\Gamma) \vdash h : \mathsf{realizes}\ \hat{h}\ \varphi$, as required.

Case $q = (\boldsymbol{\lambda} h : P.\ p)$ Let $\varphi = P \Longrightarrow Q$. If $\mathsf{typeof}\ P = \tau \neq \mathsf{Null}$ and $\mathsf{typeof}\ Q \neq \mathsf{Null}$, then $(\mathcal{R}(\Gamma), \hat{h} :: \tau, h : \mathsf{realizes}\ \hat{h}\ P) \vdash \mathsf{corr}\ (\Gamma, h : P)\ p : \mathsf{realizes}\ (\mathsf{extr}\ (\Gamma, h : P)\ p)\ Q$ by induction hypothesis. Hence

$$\begin{array}{l}
\mathcal{R}(\Gamma) \vdash \boldsymbol{\lambda}(\hat{h} :: \tau)\ (h : \mathsf{realizes}\ \hat{h}\ P).\ \mathsf{corr}\ (\Gamma, h : P)\ p : \\
\quad \bigwedge \hat{h} :: \tau.\ \mathsf{realizes}\ \hat{h}\ P \Longrightarrow \mathsf{realizes}\ (\mathsf{extr}\ (\Gamma, h : P)\ p)\ Q
\end{array}$$

and therefore

$$\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ (\boldsymbol{\lambda} h : P.\ p) : \mathsf{realizes}\ (\mathsf{extr}\ \Gamma\ (\boldsymbol{\lambda} h : P.\ p))\ (P \Longrightarrow Q)$$

as required. The other three subcases are similar.

Case $q = (\boldsymbol{\lambda} x :: \tau.\ p)$ Let $\varphi = (\bigwedge x :: \tau.\ P)$. If $\mathsf{typeof}\ P \neq \mathsf{Null}$, then $\mathcal{R}(\Gamma, x :: \tau) \vdash \mathsf{corr}\ (\Gamma, x :: \tau)\ p : \mathsf{realizes}\ (\mathsf{extr}\ (\Gamma, x :: \tau)\ p)\ P$ by induction hypothesis. Hence

$$\mathcal{R}(\Gamma) \vdash \boldsymbol{\lambda} x :: \tau.\ \mathsf{corr}\ (\Gamma, x :: \tau)\ p : \bigwedge x :: \tau.\ \mathsf{realizes}\ (\mathsf{extr}\ (\Gamma, x :: \tau)\ p)\ P$$

and therefore

$$\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ (\boldsymbol{\lambda} x :: \tau.\ p) : \mathsf{realizes}\ (\mathsf{extr}\ \Gamma\ (\boldsymbol{\lambda} x :: \tau.\ p))\ (\bigwedge x :: \tau.\ P)$$

as required. The subcase for $\mathsf{typeof}\ P = \mathsf{Null}$ is similar.

Case $q = p_1 \cdot p_2$ Let $\varphi = Q$ where $\Gamma \vdash p_1 : P \Longrightarrow Q$ and $\Gamma \vdash p_2 : P$. If $\mathsf{typeof}\ P = \tau \neq \mathsf{Null}$ and $\mathsf{typeof}\ Q \neq \mathsf{Null}$, then $\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ p_1 : \mathsf{realizes}\ (\mathsf{extr}\ \Gamma\ p_1)\ (P \Longrightarrow Q)$ and $\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ p_2 : \mathsf{realizes}\ (\mathsf{extr}\ \Gamma\ p_2)\ Q$ by induction hypothesis. The former is equivalent to $\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ p_1 : \bigwedge x :: \tau.\ \mathsf{realizes}\ x\ P \Longrightarrow \mathsf{realizes}\ ((\mathsf{extr}\ \Gamma\ p_1)\ x)\ Q$. Hence

$$\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ p_1 \cdot (\mathsf{extr}\ \Gamma\ p_1) \bullet \mathsf{corr}\ \Gamma\ p_2 : \mathsf{realizes}\ ((\mathsf{extr}\ \Gamma\ p_1)\ (\mathsf{extr}\ \Gamma\ p_2))\ Q$$

and therefore

$$\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ (p_1 \bullet p_2) : \mathsf{realizes}\ (\mathsf{extr}\ \Gamma\ (p_1 \bullet p_2))\ Q$$

as required. The other subcases are similar.

Case $q = p \cdot t$ Let $\varphi = P\ t$ where $\Gamma \vdash p : \bigwedge x.\ P\ x$ and $\mathsf{typeof}\ P \neq \mathsf{Null}$. By induction hypothesis, we have $\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ p : \mathsf{realizes}\ (\mathsf{extr}\ \Gamma\ p)\ (\bigwedge x.\ P\ x)$, which is equivalent to $\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ p : \bigwedge x.\ \mathsf{realizes}\ ((\mathsf{extr}\ \Gamma\ p)\ x)\ (P\ x)$. Hence

$$\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ p \cdot t : \mathsf{realizes}\ ((\mathsf{extr}\ \Gamma\ p)\ t)\ (P\ t)$$

due to the substitution property, and therefore

$$\mathcal{R}(\Gamma) \vdash \mathsf{corr}\ \Gamma\ (p \cdot t) : \mathsf{realizes}\ (\mathsf{extr}\ \Gamma\ (p\ t))\ (P\ t)$$

as required. The subcase for $\mathsf{typeof}\ P = \mathsf{Null}$ is similar.

Case $q = c_{\{\overline{\alpha} \mapsto \overline{\tau}\}} \cdot \overline{t}$ Let $\Sigma(c) = \bigwedge \overline{x}.\ \psi$ and $\varphi = \psi\{\overline{\alpha} \mapsto \overline{\tau},\ \overline{x} \mapsto \overline{t}\}$. Hence the claim

$$\begin{array}{l}\mathcal{R}(\Gamma) \vdash \mathcal{C}(c, \mathsf{RVars}\ c\ \overline{t})(\{\overline{\alpha} \mapsto \overline{\tau}\} \cup \mathsf{TInst}\ c\ \overline{t}) \cdot \mathcal{R}_{\mathsf{args}}(\overline{t}) : \\ \quad \mathsf{realizes}\ (\mathcal{E}(c, \mathsf{RVars}\ c\ \overline{t})(\{\overline{\alpha} \mapsto \overline{\tau}\} \cup \mathsf{TInst}\ c\ \overline{t})\ (\mathcal{E}_{\mathsf{args}}(\overline{t})))\ (\psi\{\overline{\alpha} \mapsto \overline{\tau},\ \overline{x} \mapsto \overline{t}\})\end{array}$$

follows from the assumption about $\mathcal{C}$ and $\mathcal{E}$ together with the substitution property.

4.2.4 Limitations

It is important to note that the type extraction scheme outlined in §4.2.1 is not applicable to all formulae of higher order logic in an unrestricted way. We briefly discuss two restrictions which have to be taken into account when writing specifications intended for program extraction.

Impredicativity Predicate variables in a specification become type variables in the type of the program extracted from the proof of the specification. Consequently, quantification over predicate variables in the specification corresponds to quantification over type variables in the type of the extracted program. Isabelle/Pure offers schematic polymorphism à la Hindley and Milner, where type variables are considered to be *implicitly* quantified at the outermost level, but there is no way to *explicitly* abstract or quantify over type variables. Thus, in order for the extracted program to be typable in Isabelle/Pure, predicate variables in specifications may only be quantified at the outermost level, too. This rules out specifications involving unrestricted *impredicative* quantification, such as in

$$wf\ R \ \equiv\ (\bigwedge P.\ (\bigwedge x.\ (\bigwedge y.\ R\ y\ x \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow (\bigwedge x.\ P\ x))$$

which characterizes the well-foundedness of the relation R. To see why this is problematic, imagine a proof of a theorem $wf\ R \Longrightarrow \varphi$, in which the assumption $wf\ R$ is used to establish several different computationally relevant propositions by *well-founded induction*. Hence, in the function of type $\mathsf{typeof}\ (wf\ R) \Rightarrow \mathsf{typeof}\ \varphi$ extracted from this proof, the type variable α_P corresponding to the quantified predicate variable P in $wf\ R$ would have to be instantiated to several different types, which is impossible with Hindley-Milner polymorphism.

Fortunately, impredicative specifications can often be rephrased using inductive definitions, which have a natural computational interpretation. For example, instead of the predicate wf given above, one could as well use an inductive characterization of the *accessible part* of the relation R (§4.3.5.3).

It has been shown by Paulin-Mohring [85, 86] that programs extracted from proofs in the pure Calculus of Constructions [28], which is impredicative, too, are only typable in systems that are at least as expressive as F_ω. This also makes it difficult to use standard functional programming languages such as ML as a target language for program extraction from proofs in the pure Calculus of Constructions. According to Werner [121, §1.6.1], this observation was one of the main reasons for the introduction of inductive definitions as a primitive concept into the Calculus of Constructions.

Strong eliminations Some object logics allow for the definition of predicates by recursion over some inductive datatype. In type theory jargon, such a construction is referred to as a *strong elimination* [87, §3.2.2]. For example, in Isabelle/HOL we can write

$\mathsf{datatype}\ dt\ =\ C_1\ \cdots\ |\ C_2\ \cdots$

$\mathsf{consts}\ p :: dt \Rightarrow \mathsf{bool}$

$\mathsf{primrec}$
$p\ (C_1\ \cdots) = \varphi$
$p\ (C_2\ \cdots) = \psi$

This is problematic if the formula $p\ x$ is supposed to have a computational content, since the type extracted from it *depends* on the value of x, which cannot be expressed in a target language without dependent types.

4.3 Program extraction for Isabelle/HOL

So far, we have presented a generic framework for program extraction. We will now show how to instantiate it to a specific object logic, namely Isabelle/HOL. This is done by giving additional clauses for the functions typeof, $\mathsf{realizes}$, $\mathcal{E}$ and $\mathcal{C}$. For the security conscious user, it may be reassuring to know that the introduction of new clauses cannot give rise to unsound theorems. More precisely, it is impossible to produce "wrong" correctness theorems falsely claiming that a program satisfies its specification. This is due to the architecture of the program extraction framework outlined in §4.1, which always requires correctness proofs to be verified by the proof checker.

4.3.1 Type extraction

First of all, we need to assign types to logical formulae of HOL, i.e. add new equations characterizing typeof.

$$\mathsf{typeof}\ (\mathsf{Tr}\ P) \equiv \mathsf{typeof}\ P$$

$$(\bigwedge x.\ \mathsf{typeof}\ (P\ x) \equiv \mathsf{Null}) \Longrightarrow \mathsf{typeof}\ (\exists x :: \alpha.\ P\ x) \equiv \alpha$$

$$(\bigwedge x.\ \mathsf{typeof}\ (P\ x) \equiv \tau) \Longrightarrow \mathsf{typeof}\ (\exists x :: \alpha.\ P\ x) \equiv (\alpha \times \tau)$$

$$\mathsf{typeof}\ P \equiv \mathsf{Null} \Longrightarrow \mathsf{typeof}\ Q \equiv \mathsf{Null} \Longrightarrow \mathsf{typeof}\ (P \vee Q) \equiv \mathsf{sumbool}$$

$$\mathsf{typeof}\ P \equiv \mathsf{Null} \Longrightarrow \mathsf{typeof}\ Q \equiv \tau \Longrightarrow \mathsf{typeof}\ (P \vee Q) \equiv \tau\ \mathsf{option}$$

$$\mathsf{typeof}\ P \equiv \sigma \Longrightarrow \mathsf{typeof}\ Q \equiv \mathsf{Null} \Longrightarrow \mathsf{typeof}\ (P \vee Q) \equiv \sigma\ \mathsf{option}$$

$$\mathsf{typeof}\ P \equiv \sigma \Longrightarrow \mathsf{typeof}\ Q \equiv \tau \Longrightarrow \mathsf{typeof}\ (P \vee Q) \equiv (\sigma + \tau)$$

$$\mathsf{typeof}\ A \equiv \mathsf{Null} \qquad \text{if } A \text{ atomic, i.e. } A \in \{x = y,\ \mathsf{True},\ \mathsf{False},\ \ldots\}$$

We only show the equations for $\exists$ and $\vee$. The equations for $\wedge$ are quite similar and those for $\forall$ and $\longrightarrow$ look almost the same as their meta level counterparts introduced in 4.2.1. The first equation states that typeof can simply be pushed through the coercion function Tr. The computational content of $\exists x.\ P\ x$ is either a pair consisting of the witness and the computational content of $P\ x$, if there is one, otherwise it is just the witness. If both P and Q have a computational content, then the computational content of $P \vee Q$ is a disjoint sum

$$\mathsf{datatype}\ (\alpha + \beta)\ =\ \mathit{Inl}\ \alpha \mid \mathit{Inr}\ \beta$$

If just one of P and Q has a computational content, the result is of type

$$\mathsf{datatype}\ \alpha\ \mathit{option}\ =\ \mathit{None} \mid \mathit{Some}\ \alpha$$

i.e. a program satisfying this specification will either return a proper value or signal an error. If neither P nor Q has a computational content, the result is just a boolean value, i.e. an element of type

$$\mathsf{datatype}\ \mathit{sumbool}\ =\ \mathit{Left} \mid \mathit{Right}$$

4.3.2 Realizability

In order to reason about correctness of programs extracted from HOL proofs, we also need to add equations for $\mathsf{realizes}$.

$$\mathsf{realizes}\ t\ (\mathsf{Tr}\ P) \equiv \mathsf{Tr}\ (\mathsf{realizes}\ t\ P)$$

$$(\bigwedge x.\ \mathsf{typeof}\ (P\ x) \equiv \mathsf{Null}) \Longrightarrow \mathsf{realizes}\ t\ (\exists x.\ P\ x) \equiv \mathsf{realizes}\ \mathsf{Null}\ (P\ t)$$

$$\mathsf{realizes}\ t\ (\exists x.\ P\ x) \equiv \mathsf{realizes}\ (\mathit{snd}\ t)\ (P\ (\mathit{fst}\ t))$$

$$\begin{array}{l}\mathsf{typeof}\ P \equiv \mathsf{Null} \Longrightarrow \mathsf{typeof}\ Q \equiv \mathsf{Null} \Longrightarrow \\ \quad \mathsf{realizes}\ t\ (P \vee Q)) \equiv (\mathit{case}\ t\ \mathit{of}\ \ \mathit{Left} \Rightarrow \mathsf{realizes}\ \mathsf{Null}\ P \mid \mathit{Right} \Rightarrow \mathsf{realizes}\ \mathsf{Null}\ Q)\end{array}$$

$$\begin{array}{l}\mathsf{typeof}\ P \equiv \mathsf{Null} \Longrightarrow \\ \quad \mathsf{realizes}\ t\ (P \vee Q) \equiv (\mathit{case}\ t\ \mathit{of}\ \ \mathit{None} \Rightarrow \mathsf{realizes}\ \mathsf{Null}\ P \mid \mathit{Some}\ q \Rightarrow \mathsf{realizes}\ q\ Q)\end{array}$$

$$\begin{array}{l}\mathsf{typeof}\ Q \equiv \mathsf{Null} \Longrightarrow \\ \quad \mathsf{realizes}\ t\ (P \vee Q) \equiv (\mathit{case}\ t\ \mathit{of}\ \ \mathit{None} \Rightarrow \mathsf{realizes}\ \mathsf{Null}\ Q \mid \mathit{Some}\ p \Rightarrow \mathsf{realizes}\ p\ P)\end{array}$$

$$\mathsf{realizes}\ t\ (P \vee Q) \equiv (\mathit{case}\ t\ \mathit{of}\ \ \mathit{Inl}\ p \Rightarrow \mathsf{realizes}\ p\ P \mid \mathit{Inr}\ q \Rightarrow \mathsf{realizes}\ q\ Q)$$

Again, the equations for $\wedge$ are similar and those for $\forall$ and $\longrightarrow$ look almost the same as their meta level counterparts from 4.2.3. For atomic predicates A, we set $\mathsf{realizes\ Null}\ A = A$. The above characterization of $\mathsf{realizes}$ can be applied to $\neg$ as follows: Let $\mathsf{typeof}\ P = \tau$ and $\tau \neq \mathsf{Null}$. Then

$$\begin{array}{lll} & \mathsf{realizes\ Null}\ (\neg P) & \\ = & \mathsf{realizes\ Null}\ (P \longrightarrow \mathsf{False}) & \{\text{definition of } \neg\} \\ = & \forall x :: \tau.\ \mathsf{realizes}\ x\ P \longrightarrow \mathsf{realizes\ Null\ False} & \{\text{definition of } \mathsf{realizes}\} \\ = & \forall x :: \tau.\ \mathsf{realizes}\ x\ P \longrightarrow \mathsf{False} & \{\text{definition of } \mathsf{realizes}\} \\ = & \forall x :: \tau.\ \neg \mathsf{realizes}\ x\ P & \{\text{definition of } \neg\} \end{array}$$

If $\tau = \mathsf{Null}$, then $\mathsf{realizes\ Null}\ (\neg P)$ is simply $\neg\mathsf{realizes\ Null}\ P$. Note that for P without computational content, we do not necessarily have $\mathsf{realizes\ Null}\ P = P$, but only if P contains neither $\exists$ nor $\vee$. For example, $\mathsf{realizes\ Null}\ (\neg(\exists x.\ x = c)) = \forall x.\ \neg x = c$.

4.3.3 Realizing terms

What remains to do is to specify how the functions $\mathcal{E}$ and $\mathcal{C}$ introduced in 4.2.2 and 4.2.3 act on theorems of Isabelle/HOL. This means that for each basic inference rule of the logic, we have to give a realizing term and a correctness proof. As before, we only treat some particularly interesting cases. Figure 4.2 shows the realizing terms corresponding to some of the inference rules of HOL. As mentioned in §4.2.1, there may be more than one realizer for each inference rule. The correctness of these programs, i.e. the fact that they realize the corresponding inference rules, follows quite easily from the basic properties of the functions and datatypes involved. For example, the correctness of the programs extracted from rule *disjE*

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

is expressed by the theorems[1]

$$\begin{array}{l} \mathit{case}\ x\ \mathit{of}\ \mathit{Inl}\ p \Rightarrow P^R\ p \mid \mathit{Inr}\ q \Rightarrow Q^R\ q \Longrightarrow \\ (\bigwedge p.\ P^R\ p \Longrightarrow R^R\ (f\ p)) \Longrightarrow \\ (\bigwedge q.\ Q^R\ q \Longrightarrow R^R\ (g\ q)) \Longrightarrow R^R\ (\mathit{case}\ x\ \mathit{of}\ \mathit{Inl}\ p \Rightarrow f\ p \mid \mathit{Inr}\ q \Rightarrow g\ q) \end{array}$$

$$\begin{array}{l} \mathit{case}\ x\ \mathit{of}\ \mathit{None} \Rightarrow P^R \mid \mathit{Some}\ q \Rightarrow Q^R\ q \Longrightarrow \\ (P^R \Longrightarrow R^R\ f) \Longrightarrow \\ (\bigwedge q.\ Q^R\ q \Longrightarrow R^R\ (g\ q)) \Longrightarrow R^R\ (\mathit{case}\ x\ \mathit{of}\ \mathit{None} \Rightarrow f \mid \mathit{Some}\ q \Rightarrow g\ q) \end{array}$$

$$\begin{array}{l} \mathit{case}\ x\ \mathit{of}\ \mathit{Left} \Rightarrow P^R \mid \mathit{Right} \Rightarrow Q^R \Longrightarrow \\ (P^R \Longrightarrow R^R\ f) \Longrightarrow (Q^R \Longrightarrow R^R\ g) \Longrightarrow R^R\ (\mathit{case}\ x\ \mathit{of}\ \mathit{Left} \Rightarrow f \mid \mathit{Right} \Rightarrow g) \end{array}$$

which correspond to the cases where the set of computationally relevant variables is $\{P,\ Q,\ R\}$, $\{Q,\ R\}$ or $\{R\}$, respectively. The above theorems can be proved by case analysis on x followed by an application of the rewrite rules for the case combinators involved. Similarly, the correctness theorem for the program extracted from rule *exI*

$$P\ x \Longrightarrow \exists x.\ P\ x$$

for the case where the variable P is computationally relevant is expressed by

[1]Note that we actually show the *Harrop normal form* of the correctness statements, i.e. we write e.g. $P\ x \Longrightarrow Q\ x\ y \Longrightarrow R\ x\ y$ instead of $\bigwedge x.\ P\ x \Longrightarrow (\bigwedge y.\ Q\ x\ y \Longrightarrow R\ x\ y)$.

name	V	$\mathcal{E}(name,\ V)$
impI	$\{P,\ Q\}$	$\lambda pq.\ pq$
	$\{Q\}$	$\lambda q.\ q$
mp	$\{P,\ Q\}$	$\lambda pq.\ pq$
	$\{Q\}$	$\lambda q.\ q$
allI	$\{P\}$	$\lambda p.\ p$
spec	$\{P\}$	$\lambda x\ p.\ p\ x$
exI	$\{P\}$	$\lambda x\ y.\ (x,\ y)$
	$\{\}$	$\lambda x.\ x$
exE	$\{P,\ Q\}$	$\lambda p\ pq.\ case\ p\ of\ (x,\ y) \Rightarrow pq\ x\ y$
	$\{Q\}$	$\lambda x\ pq.\ pq\ x$
conjI	$\{P,\ Q\}$	$\lambda x\ p.\ (x,\ p)$
	$\{P\}$	$\lambda p.\ p$
	$\{Q\}$	$\lambda q.\ q$
conjunct1	$\{P,\ Q\}$	*fst*
	$\{P\}$	$\lambda p.\ p$
conjunct2	$\{P,\ Q\}$	*snd*
	$\{Q\}$	$\lambda p.\ p$
disjI1	$\{P,\ Q\}$	*Inl*
	$\{P\}$	*Some*
	$\{Q\}$	*None*
	$\{\}$	*Left*
disjI2	$\{P,\ Q\}$	*Inr*
	$\{P\}$	*None*
	$\{Q\}$	*Some*
	$\{\}$	*Right*
disjE	$\{P,\ Q,\ R\}$	$\lambda pq\ pr\ qr.\ case\ pq\ of\ Inl\ p \Rightarrow pr\ p \mid Inr\ q \Rightarrow qr\ q$
	$\{Q,\ R\}$	$\lambda pq\ pr\ qr.\ case\ pq\ of\ None \Rightarrow pr \mid Some\ q \Rightarrow qr\ q$
	$\{P,\ R\}$	$\lambda pq\ pr\ qr.\ case\ pq\ of\ None \Rightarrow qr \mid Some\ p \Rightarrow pr\ p$
	$\{R\}$	$\lambda pq\ pr\ qr.\ case\ pq\ of\ Left \Rightarrow pr \mid Right \Rightarrow qr$
FalseE	$\{P\}$	*arbitrary*
subst	$\{P\}$	$\lambda s\ t\ ps.\ ps$

Figure 4.2: Realizers for basic inference rules of Isabelle/HOL

$$P^R\ y\ x \Longrightarrow P^R\ (snd\ (x,\ y))\ (fst\ (x,\ y))$$

where x is the existential witness and y is the computational content of $P\ x$. For the program corresponding to the existential elimination rule *exE*

$$\exists x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q) \Longrightarrow Q$$

the correctness theorem is

$$P^R\ (snd\ p)\ (fst\ p) \Longrightarrow$$
$$(\bigwedge x\ y.\ P^R\ y\ x \Longrightarrow Q^R\ (f\ x\ y)) \Longrightarrow Q^R\ (case\ p\ of\ (x,\ y) \Rightarrow f\ x\ y)$$

Note that in order to avoid unnecessary duplication of the term p in the extracted program, the computational content of *exE* is expressed using the *case* operator for pairs instead of *fst* and *snd*. The program corresponding to the elimination rule *FalseE*

$$False \Longrightarrow P$$

may look a bit peculiar at first sight. However, since we may prove anything in a context containing the assumption *False*, we may in particular prove that any program realizes the specification P, which is why we take the default constant *arbitrary* as a realizer for *FalseE*. Finally, we turn to the computational content of the substitution rule

$$s = t \Longrightarrow P\ s \Longrightarrow P\ t$$

Intuitively, applying a substitution should not change the computational content. For example, it should be possible to extract the same program from a proof of $\exists\, y.\ P\ (x + \mathit{0})\ y$ as one can extract from a proof of $\exists\, y.\ P\ x\ y$. Therefore, the substitution rule is realized by the *identity function*. To see why this is correct, recall that

$$\begin{array}{ll} & \mathsf{realizes}\ (\lambda ps.\ ps)\ (s = t \Longrightarrow P\ s \Longrightarrow P\ t) \\ = & \mathsf{realizes\ Null}\ (s = t) \Longrightarrow \mathsf{realizes}\ (\lambda ps.\ ps)\ (P\ s \Longrightarrow P\ t) \\ = & \mathsf{realizes\ Null}\ (s = t) \Longrightarrow (\bigwedge r.\ \mathsf{realizes}\ r\ (P\ s) \Longrightarrow \mathsf{realizes}\ r\ (P\ t)) \\ = & s = t \Longrightarrow (\bigwedge r.\ P^R\ r\ s \Longrightarrow P^R\ r\ t) \end{array}$$

i.e. the correctness statement is again an instance of the substitution rule itself. However, when using the substitution rule in proofs from which one wants to extract programs, some care is required: Since in HOL, the type *bool* is a type like any other, the substitution rule may also be used if s and t are of type *bool*. Since $s = t$ is identified with $(s \longrightarrow t) \wedge (t \longrightarrow s)$ for s and t of type *bool*, equality on booleans actually has a computational content itself. For example, a proof of $(A \wedge B) = (B \wedge A)$ would correspond to the program

$$(\lambda ab.\ (\mathit{snd}\ ab,\ \mathit{fst}\ ab),\ \lambda ba.\ (\mathit{snd}\ ba,\ \mathit{fst}\ ba))$$

Moreover, applying a boolean substitution would *change* the computational content: A substitution of $B \wedge A$ for $A \wedge B$ in the context $\lambda X.\ X \wedge C$ would correspond to the program

$$\lambda abc.\ ((\mathit{snd}\ (\mathit{fst}\ abc),\ \mathit{fst}\ (\mathit{fst}\ abc)),\ \mathit{snd}\ abc)$$

In particular, finding out the program corresponding to such a substitution would also require a rather cumbersome analysis of the substitution context (denoted by P in the above rule) that would not fit nicely into the program extraction framework presented in §4.2. We therefore stipulate that proofs to which the functions extr and corr are applied, do not contain any boolean substitutions. These have to be replaced by suitable *congruence rules* for logical operators in a preprocessing step prior to the invocation of the extraction function.

4.3.4 Realizers for inductive datatypes

4.3.4.1 Introduction

So far we have only covered the computational content of *basic* inference rules. However, just the basic inference rules alone are often insufficient for practical applications. An important proof principle which is frequently used in realistic proofs is *structural induction* on datatypes. This section is concerned with a definition of the computational content of such proofs. Although we will review the basic properties of datatypes, as far as they are relevant for program extraction, we will not discuss the actual *construction* of such datatypes in HOL by means of *fixpoint operators*. The interested reader may find the details e.g. in the articles by Paulson [93] as well as Berghofer and Wenzel [21].

As an introductory example, consider the datatype

$$\mathsf{datatype}\ nat \ = \ 0 \mid Suc\ nat$$

of *natural numbers*. The induction principle for this datatype is

$$P\ 0 \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n)) \Longrightarrow P\ n$$

According to the definition of type extraction given in §4.2.1, a program corresponding to this rule must have the type $\alpha_P \Rightarrow (nat \Rightarrow \alpha_P \Rightarrow \alpha_P) \Rightarrow \alpha_P$, which is the type of the recursion combinator

$$\begin{array}{l} nat\text{-}rec\ f\ g\ 0 = f \\ nat\text{-}rec\ f\ g\ (Suc\ nat) = g\ nat\ (nat\text{-}rec\ f\ g\ nat) \end{array}$$

To see that *nat-rec* is indeed a correct realizer for the induction rule on natural numbers, note that

$$\begin{array}{ll} & \mathsf{realizes}\ (\lambda f\ g.\ nat\text{-}rec\ f\ g\ n)\ (P\ 0 \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n)) \Longrightarrow P\ n) \\ = & \bigwedge f.\ P^R\ f\ 0 \Longrightarrow (\bigwedge g.\ (\bigwedge n\ x.\ P^R\ x\ n \Longrightarrow P^R\ (g\ n\ x)\ (Suc\ n)) \Longrightarrow P^R\ (nat\text{-}rec\ f\ g\ n)\ n) \end{array}$$

as has already been mentioned in §4.2.3. This correctness statement can easily be proved as follows. Assume we have suitable realizers f and g for the induction basis and induction step, respectively, i.e. the corresponding correctness statements are $P^R\ f\ 0$ and $\bigwedge n\ x.\ P^R\ x\ n \Longrightarrow P^R\ (g\ n\ x)\ (Suc\ n)$. By induction on n, we then show that $P^R\ (nat\text{-}rec\ f\ g\ n)\ n$. For the induction basis, we need to show $P^R\ (nat\text{-}rec\ f\ g\ 0)\ 0$, which easily follows from the correctness statement for f, together with the characteristic equations for *nat-rec*. For the induction step, we may already assume that $P\ (nat\text{-}rec\ f\ g\ n)\ n$ holds for some n. Because of the correctness statement for g, this implies $P^R\ (g\ n\ (nat\text{-}rec\ f\ g\ n))\ (Suc\ n)$ and therefore $P^R\ (nat\text{-}rec\ f\ g\ (Suc\ n))\ (Suc\ n)$, again due to the characteristic equations for *nat-rec*, which concludes the proof. This proof can easily be expressed in Isabelle/Isar as follows:

theorem *nat-ind-correctness*:
  assumes *r0*: $P^R\ f\ 0$
  and *rSuc*: $\bigwedge n\ x.\ P^R\ x\ n \Longrightarrow P^R\ (g\ n\ x)\ (Suc\ n)$
  shows $P^R\ (nat\text{-}rec\ f\ g\ n)\ n$
proof (*induct n*)
  from *r0* **show** $P^R\ (nat\text{-}rec\ f\ g\ 0)\ 0$ **by** *simp*
next
  fix n **assume** $P^R\ (nat\text{-}rec\ f\ g\ n)\ n$
  hence $P^R\ (g\ n\ (nat\text{-}rec\ f\ g\ n))\ (Suc\ n)$ **by** (*rule rSuc*)
  thus $P^R\ (nat\text{-}rec\ f\ g\ (Suc\ n))\ (Suc\ n)$ **by** *simp*
qed

Similarly, the program corresponding to the weaker *case analysis* theorem

$$(y = 0 \Longrightarrow P) \Longrightarrow (\bigwedge nat.\ y = Suc\ nat \Longrightarrow P) \Longrightarrow P$$

must have type $\alpha_P \Rightarrow (nat \Rightarrow \alpha_P) \Rightarrow \alpha_P$, which is the type of the case combinator

$$\begin{array}{l} nat\text{-}case\ f\ g\ 0 = f \\ nat\text{-}case\ f\ g\ (Suc\ nat) = g\ nat \end{array}$$

By an argument similar to the one used for the induction theorem above, we can show that *nat-case* is a suitable realizer for the case analysis theorem, i.e.

$$\begin{array}{rl} & \textsf{realizes}\ (\lambda f\ g.\ \textit{nat-case}\ f\ g\ n)\ ((n = 0 \Longrightarrow P) \Longrightarrow (\bigwedge n'.\ n = \textit{Suc}\ n' \Longrightarrow P) \Longrightarrow P) \\ = & \bigwedge f.\ (n = 0 \Longrightarrow P^R\ f) \Longrightarrow \\ & \quad (\bigwedge g.\ (\bigwedge n'.\ n = \textit{Suc}\ n' \Longrightarrow P^R\ (g\ n')) \Longrightarrow P^R\ (\textit{nat-case}\ f\ g\ n)) \end{array}$$

This correctness statement is easily proved by a case analysis on n, which can be phrased in Isabelle/Isar as follows

theorem *nat-case-correctness*:
  assumes *r0*: $n = 0 \Longrightarrow P^R\ f$
  and *rSuc*: $\bigwedge n'.\ n = \textit{Suc}\ n' \Longrightarrow P^R\ (g\ n')$
  shows $P^R\ (\textit{nat-case}\ f\ g\ n)$
proof (*cases n*)
  assume $n = 0$
  thus $P^R\ (\textit{nat-case}\ f\ g\ n)$ **by** *simp* (*rule r0*)
next
  fix n' **assume** $n = \textit{Suc}\ n'$
  thus $P^R\ (\textit{nat-case}\ f\ g\ n)$ **by** *simp* (*rule rSuc*)
qed

4.3.4.2 General scheme

We will now generalize what we have just explained with an example. Consider the general case of a datatype definition

$$\begin{array}{llcl} \textsf{datatype} & \overline{\alpha}\ t_1 & = & C^1_1\ \tau^1_{1,1}\ \ldots\ \tau^1_{1,m^1_1}\ \mid\ \ldots\ \mid\ C^1_{k_1}\ \tau^1_{k_1,1}\ \ldots\ \tau^1_{k_1,m^1_{k_1}} \\ & & \vdots & \\ \textsf{and} & \overline{\alpha}\ t_n & = & C^n_1\ \tau^n_{1,1}\ \ldots\ \tau^n_{1,m^n_1}\ \mid\ \ldots\ \mid\ C^n_{k_n}\ \tau^n_{k_n,1}\ \ldots\ \tau^n_{k_n,m^n_{k_n}} \end{array}$$

We call a type argument $\tau^j_{i,i'}$ *recursive*, if it contains any of the newly defined type constructors $t_1, \ldots, t_n$, otherwise *nonrecursive*. We denote by $r^j_{i,1}, \ldots, r^j_{i,l^j_i}$ the positions of the recursive arguments of the i-th constructor of the j-th datatype. A recursive type argument $\tau^j_{i,r^j_{i,l}}$ must have the form $\overline{\sigma^j_{i,l}} \Rightarrow \overline{\alpha}\ t_{s^j_{i,l}}$, where $1 \leq l \leq l^j_i$ and $\overline{\sigma^j_{i,l}}$ does *not* contain any of the newly defined type constructors. This means that $t_1, \ldots, t_n$ may only occur *strictly positive* in $\tau^j_{i,i'}$.

Induction The rule for simultaneous structural induction on the types $\overline{\alpha}\ t_1$, …, $\overline{\alpha}\ t_n$ has the form

$$\mathcal{I}^1_1 \Longrightarrow \cdots \Longrightarrow \mathcal{I}^1_{k_1} \Longrightarrow \cdots \Longrightarrow \mathcal{I}^n_1 \Longrightarrow \cdots \Longrightarrow \mathcal{I}^n_{k_n} \Longrightarrow P_1\ x_1 \wedge \ldots \wedge P_n\ x_n$$

where

$$\mathcal{I}^j_i = \bigwedge \overline{x^j_i}.\ \left(\bigwedge \overline{z^j_{i,1}}.\ P_{s^j_{i,1}}\left(x_{r^j_{i,1}}\ \overline{z^j_{i,1}}\right)\right) \Longrightarrow \cdots \Longrightarrow \left(\bigwedge \overline{z^j_{i,l^j_i}}.\ P_{s^j_{i,l^j_i}}\left(x_{r^j_{i,l^j_i}}\ \overline{z^j_{i,l^j_i}}\right)\right) \Longrightarrow P_j\left(C^j_i\ \overline{x^j_i}\right)$$

$$\overline{x^j_i} = x_1 \ldots x_{m^j_i}$$

Case analysis The rule for case analysis on the type $\overline{\alpha}\ t_j$ has the form

$$\left(\bigwedge \overline{x^j_1}.\ y = C^j_1\ \overline{x^j_1} \Longrightarrow P\right) \Longrightarrow \cdots \Longrightarrow \left(\bigwedge \overline{x^j_{k_j}}.\ y = C^j_{k_j}\ \overline{x^j_{k_j}} \Longrightarrow P\right) \Longrightarrow P$$

Recursion The combinators t_1-*rec*, …, t_n-*rec* for mutual recursion on the types $\overline{\alpha}\ t_1$, …, $\overline{\alpha}\ t_n$ are characterized by the equations

$$t_j\textit{-rec}\ f_1^1 \dots f_{k_n}^n\ \left(C_i^j\ \overline{x_i^j}\right) = f_i^j\ \overline{x_i^j}\ p_1 \dots p_{l_i^j}$$

where

$$p_l = \lambda \overline{z_{i,l}^j}.\ t_{s_{i,l}^j}\textit{-rec}\ f_1^1 \dots f_{k_n}^n\ \left(x_{r_{i,l}^j}\ \overline{z_{i,l}^j}\right)$$

Realizer for induction rule To simplify the presentation, we assume that all of the predicates P_1, …, P_n have a computational content. Then, the above simultaneous induction rule is realized by the term

$$\lambda f_1^1 \dots f_{k_n}^n.\ \left(t_1\textit{-rec}\ f_1^1 \dots f_{k_n}^n\ x_1,\ \dots,\ t_n\textit{-rec}\ f_1^1 \dots f_{k_n}^n\ x_n\right)$$

The fact that this term is a correct realizer can be expressed as follows

$$\begin{array}{l} \mathcal{R}_1^1\ f_1^1 \Longrightarrow \dots \Longrightarrow \mathcal{R}_{k_1}^1\ f_{k_1}^1 \Longrightarrow \dots \Longrightarrow \mathcal{R}_1^n\ f_1^n \Longrightarrow \dots \Longrightarrow \mathcal{R}_{k_n}^n\ f_{k_n}^n \Longrightarrow \\ \quad P_1^R\ \left(\textit{fst}\ \left(\cdots \left(\textit{fst}\ \left(t_1\textit{-rec}\ f_1^1 \dots f_{k_n}^n\ x_1,\ \dots,\ t_n\textit{-rec}\ f_1^1 \dots f_{k_n}^n\ x_n\right)\right) \cdots\right)\right)\ x_1 \wedge \dots \wedge \\ \quad P_n^R\ \left(\textit{snd}\ \left(\cdots \left(\textit{snd}\ \left(t_1\textit{-rec}\ f_1^1 \dots f_{k_n}^n\ x_1,\ \dots,\ t_n\textit{-rec}\ f_1^1 \dots f_{k_n}^n\ x_n\right)\right) \cdots\right)\right)\ x_n \end{array}$$

where

$$\begin{array}{l} \mathcal{R}_i^j\ f = \bigwedge \overline{x_i^j}\ p_1.\ \left(\bigwedge \overline{z_{i,1}^j}.\ P_{s_{i,1}^j}^R\ \left(p_1\ \overline{z_{i,1}^j}\right)\ \left(x_{r_{i,1}^j}\ \overline{z_{i,1}^j}\right)\right) \Longrightarrow \Bigg(\cdots \Longrightarrow \\ \qquad \left(\bigwedge p_{l_i^j}.\ \left(\bigwedge \overline{z_{i,l_i^j}^j}.\ P_{s_{i,l_i^j}^j}^R\ \left(p_{l_i^j}\ \overline{z_{i,l_i^j}^j}\right)\ \left(x_{r_{j,l_i^j}^i}\ \overline{z_{i,l_i^j}^j}\right)\right) \Longrightarrow P_j^R\ \left(f\ \overline{x_i^j}\ \overline{p_i^j}\right)\ \left(C_i^j\ \overline{x_i^j}\right)\right) \cdots \Bigg) \\ \overline{p_i^j} = p_1 \dots p_{l_i^j} \end{array}$$

This correctness theorem is proved by simultaneous induction on $x_1 \dots x_n$. Assume we have $\mathcal{R}_1^1\ f_1^1$, …, $\mathcal{R}_{k_n}^n\ f_{k_n}^n$. For constructor C_i^j, we obtain the induction hypotheses

$$\begin{array}{l} \bigwedge \overline{z_{i,1}^j}.\ P_{s_{i,1}^j}^R\ \left(t_{s_{i,1}^j}\textit{-rec}\ f_1^1 \dots f_{k_n}^n\ \left(x_{r_{i,1}^j}\ \overline{z_{i,1}^j}\right)\right)\ \left(x_{r_{i,1}^j}\ \overline{z_{i,1}^j}\right) \\ \qquad \vdots \\ \bigwedge \overline{z_{i,l_i^j}^j}.\ P_{s_{i,l_i^j}^j}^R\ \left(t_{s_{i,l_i^j}^j}\textit{-rec}\ f_1^1 \dots f_{k_n}^n\ \left(x_{r_{i,l_i^j}^j}\ \overline{z_{i,l_i^j}^j}\right)\right)\ \left(x_{r_{i,l_i^j}^j}\ \overline{z_{i,l_i^j}^j}\right) \end{array}$$

from which we have to show

$$P_j^R\ \left(t_j\textit{-rec}\ f_1^1 \dots f_{k_n}^n\ \left(C_i^j\ \overline{x_i^j}\right)\right)\ \left(C_i^j\ \overline{x_i^j}\right)$$

for all $\overline{x_i^j}$. This is equivalent to

$$P_j^R\ \left(f_i^j\ \overline{x_i^j}\ p_1 \dots p_{l_i^j}\right)\ \left(C_i^j\ \overline{x_i^j}\right)$$

where

$$p_l = \lambda \overline{z_{i,l}^j}.\ t_{s_{i,l}^j}\textit{-rec}\ f_1^1 \dots f_{k_n}^n\ \left(x_{r_{i,l}^j}\ \overline{z_{i,l}^j}\right)$$

which can easily be deduced from the induction hypotheses using $\mathcal{R}_i^j\ f_i^j$.

Realizer for case analysis rule The rule for case analysis on the type $\overline{\alpha}\ t_j$ is realized by the term

$$\lambda f_1 \ldots f_{k_j}.\ t_j\text{-}case\ f_1 \ldots f_{k_j}\ y$$

The correctness of this realizer is expressed by the statement

$$\mathcal{R}_1^j\ f_1\ y \Longrightarrow \cdots \Longrightarrow \mathcal{R}_{k_j}^j\ f_{k_j}\ y \Longrightarrow P^R\ (t_j\text{-}case\ f_1 \ldots f_{k_j}\ y)$$

where

$$\mathcal{R}_i^j\ f\ y = \left(\bigwedge \overline{x_i^j}.\ y = C_i^j\ \overline{x_i^j} \Longrightarrow P^R\ \left(f\ \overline{x_i^j}\right)\right)$$

This can be proved by case analysis on y. Assume we have $\mathcal{R}_1^j\ f_1\ y$, …, $\mathcal{R}_{k_j}^j\ f_{k_j}\ y$. For constructor C_i^j, we have to show

$$P^R\ \left(t_j\text{-}case\ f_1 \ldots f_{k_j}\ \left(C_i^j\ \overline{x_i^j}\right)\right)$$

for all $\overline{x_i^j}$. This is equivalent to

$$P^R\ \left(f_i\ \overline{x_i^j}\right)$$

which directly follows from $\mathcal{R}_i^j\ f_i\ y$.

4.3.5 Realizers for inductive predicates

4.3.5.1 Introduction

The concept of an *inductive predicate* is quite similar to that of an inductive datatype: While a datatype is characterized by a list of *constructors* together with their types, an inductive predicate is characterized by a list of *introduction rules.*

As an example, consider the definition of the transitive closure of a relation r, which is given by the following introduction rules:

rtrancl-refl:	$(a,\ a) \in r^*$
rtrancl-into-rtrancl:	$(a,\ b) \in r^* \Longrightarrow (b,\ c) \in r \Longrightarrow (a,\ c) \in r^*$

In order to prove that a property $P\ x\ y$ holds for all x and y with $(x,\ y) \in r^*$, one often performs induction on the *derivation* of $(x,\ y) \in r^*$, which is sometimes referred to as *rule induction.* This proof rule can be expressed as follows:

$$\begin{array}{l}(x,\ y) \in r^* \Longrightarrow \\ (\bigwedge a.\ P\ a\ a) \Longrightarrow (\bigwedge a\ b\ c.\ (a,\ b) \in r^* \Longrightarrow P\ a\ b \Longrightarrow (b,\ c) \in r \Longrightarrow P\ a\ c) \Longrightarrow P\ x\ y\end{array}$$

One may now ask which program should correspond to such a proof rule. In §4.3.4, we have seen that the program corresponding to a proof by structural induction on a datatype is a function defined by recursion on the very same datatype. Analogously, the program extracted from a proof by induction on the derivation of an inductive predicate should be a recursive function, too, where the recursion runs over a datatype which encodes the derivation. This datatype can be derived from the introduction rules in a canonical way: Each introduction rule corresponds to a constructor, whose type can be derived from the proposition of the respective introduction rule using the *type extraction* mechanism introduced in §4.2.1. We will illustrate this idea using the transitive closure predicate described above. When defining a datatype representing a derivation of $(x,\ y) \in r^*$, we need to distinguish whether or not r has a computational content. If it has no computational content, the corresponding datatype has just one type parameter α, i.e. the element type of the relation, otherwise it has one more parameter α_r, which corresponds to the type representing a derivation of $(x,\ y) \in r$. More formally,

$$(\bigwedge x\ y.\ \mathsf{typeof}\ ((x,y) \in r) \equiv \mathsf{Null}) \Longrightarrow \mathsf{typeof}\ ((x :: \alpha, y) \in r^*) \equiv \alpha\ \mathit{rtranclT}$$

$$(\bigwedge x\ y.\ \mathsf{typeof}\ ((x,y) \in r) \equiv \alpha_r) \Longrightarrow \mathsf{typeof}\ ((x :: \alpha, y) \in r^*) \equiv (\alpha_r, \alpha)\ \mathit{rtranclT\text{-}r}$$

Using the above equations for typeof, we can deduce that the types of the constructors corresponding to the introduction rules are

$$\begin{array}{ll} \mathit{rtrancl\text{-}refl} :: & \alpha \Rightarrow \alpha\ \mathit{rtranclT} \\ \mathit{rtrancl\text{-}into\text{-}rtrancl} :: & \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha\ \mathit{rtranclT} \Rightarrow \alpha\ \mathit{rtranclT} \end{array}$$

for the case where r has no computational content, and

$$\begin{array}{ll} \mathit{rtrancl\text{-}refl} :: & \alpha \Rightarrow (\alpha_r, \alpha)\ \mathit{rtranclT\text{-}r} \\ \mathit{rtrancl\text{-}into\text{-}rtrancl} :: & \alpha \Rightarrow \alpha \Rightarrow \alpha \Rightarrow (\alpha_r, \alpha)\ \mathit{rtranclT\text{-}r} \Rightarrow \alpha_r \Rightarrow (\alpha_r, \alpha)\ \mathit{rtranclT\text{-}r} \end{array}$$

for the case where r has a computational content. Note that the variables a, b and c occurring in the introduction rules are assumed to be bound by implicit universal quantifiers, which is why the above constructor functions expect one and three arguments of type α, respectively. The datatypes $\alpha\ \mathit{rtranclT}$ and $(\alpha_r, \alpha)\ \mathit{rtranclT\text{-}r}$ can therefore be defined as

$$\begin{array}{llcl} \mathsf{datatype} & \alpha\ \mathit{rtranclT} & = & \mathit{rtrancl\text{-}refl}\ \alpha \\ & & \mid & \mathit{rtrancl\text{-}into\text{-}rtrancl}\ \alpha\ \alpha\ \alpha\ (\alpha\ \mathit{rtranclT}) \end{array}$$

and

$$\begin{array}{llcl} \mathsf{datatype} & (\alpha_r, \alpha)\ \mathit{rtranclT\text{-}r} & = & \mathit{rtrancl\text{-}refl}\ \alpha \\ & & \mid & \mathit{rtrancl\text{-}into\text{-}rtrancl}\ \alpha\ \alpha\ \alpha\ ((\alpha_r, \alpha)\ \mathit{rtranclT\text{-}r})\ \alpha_r \end{array}$$

We also need to define a suitable realizability predicate, expressing that an element of the above datatype represents a derivation of $(x,\ y) \in r^*$. Like the predicate r^*, this realizability predicate will be defined inductively. While r^* is a relation with two arguments, the realizability predicate will take one more argument, which is the realizing term. Again, we need to distinguish whether or not r has a computational content. For each of the two datatypes defined above, there will be a specific realizability predicate. More formally,

$$\begin{array}{l} (\bigwedge x\ y.\ \mathsf{typeof}\ ((x,y) \in r) \equiv \mathsf{Null}) \Longrightarrow \\ \quad \mathsf{realizes}\ p\ ((x,y) \in r^*) \equiv (p,x,y) \in \mathit{rtranclR}\ (\lambda t.\ \mathsf{realizes}\ \mathsf{Null}\ (t \in r)) \end{array}$$

$$\begin{array}{l} (\bigwedge x\ y.\ \mathsf{typeof}\ ((x,y) \in r) \equiv \alpha_r) \Longrightarrow \\ \quad \mathsf{realizes}\ p\ ((x,y) \in r^*) \equiv (p,x,y) \in \mathit{rtranclR\text{-}r}\ (\lambda q\ t.\ \mathsf{realizes}\ q\ (t \in r)) \end{array}$$

Using the above equations for realizes together with the formalism introduced in §4.2.3, we can turn the introduction rules *rtrancl-refl* and *rtrancl-into-rtrancl* into rules expressing that the constructors of the datatypes *rtranclT* and *rtranclT-r* realize the respective introduction rules. These rules will serve as introduction rules for the inductive definition of the realizability predicates *rtranctR* and *rtranclR-r*, i.e. the realizability predicates will be defined to be the least[2] predicates closed under the rules presented below. If r has a computational content, we have

$$\begin{array}{ll}
 & \textsf{realizes}\ \mathit{rtranclT\text{-}r.rtrancl\text{-}refl}\ (\bigwedge a.\ (a,\ a) \in r^*) \\
= & \bigwedge a.\ \textsf{realizes}\ (\mathit{rtranclT\text{-}r.rtrancl\text{-}refl}\ a)\ ((a,\ a) \in r^*) \\
= & \bigwedge a.\ (\mathit{rtranclT\text{-}r.rtrancl\text{-}refl}\ a,\ a,\ a) \in \mathit{rtranclR\text{-}r}\ (\lambda p\ t.\ \textsf{realizes}\ p\ (t \in r)) \\
= & \bigwedge a.\ (\mathit{rtranclT\text{-}r.rtrancl\text{-}refl}\ a,\ a,\ a) \in \mathit{rtranclR\text{-}r}\ r^R
\end{array}$$

and

$$\begin{array}{ll}
 & \textsf{realizes}\ \mathit{rtranclT\text{-}r.rtrancl\text{-}into\text{-}rtrancl} \\
 & \quad (\bigwedge a\ b\ c.\ (a,\ b) \in r^* \Longrightarrow (b,\ c) \in r \Longrightarrow (a,\ c) \in r^*) \\
= & \bigwedge a\ b\ c\ p. \\
 & \quad \textsf{realizes}\ p\ ((a,\ b) \in r^*) \Longrightarrow \\
 & \quad (\bigwedge q.\ \textsf{realizes}\ q\ ((b,\ c) \in r) \Longrightarrow \\
 & \qquad\quad \textsf{realizes}\ (\mathit{rtranclT\text{-}r.rtrancl\text{-}into\text{-}rtrancl}\ a\ b\ c\ p\ q)\ ((a,\ c) \in r^*)) \\
= & \bigwedge a\ b\ c\ p. \\
 & \quad (p,\ a,\ b) \in \mathit{rtranclR\text{-}r}\ (\lambda p\ t.\ \textsf{realizes}\ p\ (t \in r)) \Longrightarrow \\
 & \quad (\bigwedge q.\ \textsf{realizes}\ q\ ((b,\ c) \in r) \Longrightarrow \\
 & \qquad\quad (\mathit{rtranclT\text{-}r.rtrancl\text{-}into\text{-}rtrancl}\ a\ b\ c\ p\ q,\ a,\ c) \\
 & \qquad\quad \in \mathit{rtranclR\text{-}r}\ (\lambda p\ t.\ \textsf{realizes}\ p\ (t \in r))) \\
= & \bigwedge a\ b\ c\ p. \\
 & \quad (p,\ a,\ b) \in \mathit{rtranclR\text{-}r}\ r^R \Longrightarrow \\
 & \quad (\bigwedge q.\ r^R\ q\ (b,\ c) \Longrightarrow \\
 & \qquad\quad (\mathit{rtranclT\text{-}r.rtrancl\text{-}into\text{-}rtrancl}\ a\ b\ c\ p\ q,\ a,\ c) \in \mathit{rtranclR\text{-}r}\ r^R)
\end{array}$$

Similarly, for r without computational content, we obtain the rules

$$\begin{array}{l}
(\mathit{rtranclT.rtrancl\text{-}refl}\ a,\ a,\ a) \in \mathit{rtranclR}\ r^R \\
(p,\ a,\ b) \in \mathit{rtranclR}\ r^R \Longrightarrow \\
r^R\ (b,\ c) \Longrightarrow (\mathit{rtranclT.rtrancl\text{-}into\text{-}rtrancl}\ a\ b\ c\ p,\ a,\ c) \in \mathit{rtranclR}\ r^R
\end{array}$$

Using this definition of realizability, we can now show that the rule for induction on the derivation of $(x,\ y) \in r^*$ is realized by the recursion combinators for the datatypes *rtranclT* and *rtranclT-r*. For simplicity, we consider the case where r has no computational content. Assume we have a realizer p for $(x,\ y) \in r^*$, i.e. $(p,\ x,\ y) \in \mathit{rtranclR}\ r^R$. Then the fact that the recursion combinator

$$\begin{array}{l}
\mathit{rtranclT\text{-}rec}\ f\ g\ (\mathit{rtranclT.rtrancl\text{-}refl}\ a) = f\ a \\
\mathit{rtranclT\text{-}rec}\ f\ g\ (\mathit{rtranclT.rtrancl\text{-}into\text{-}rtrancl}\ a\ b\ c\ p) = \\
g\ a\ b\ c\ p\ (\mathit{rtranclT\text{-}rec}\ f\ g\ p)
\end{array}$$

realizes the induction rule

$$\begin{array}{l}
(x,\ y) \in r^* \Longrightarrow \\
(\bigwedge a.\ P\ a\ a) \Longrightarrow (\bigwedge a\ b\ c.\ (a,\ b) \in r^* \Longrightarrow P\ a\ b \Longrightarrow (b,\ c) \in r \Longrightarrow P\ a\ c) \Longrightarrow P\ x\ y
\end{array}$$

can be proved by induction on the derivation of $(p,\ x,\ y) \in \mathit{rtranclR}\ r^R$, which is expressed by the rule

$$\begin{array}{l}
(p,\ x,\ y) \in \mathit{rtranclR}\ r^R \Longrightarrow \\
(\bigwedge a.\ P\ (\mathit{rtranclT.rtrancl\text{-}refl}\ a)\ a\ a) \Longrightarrow \\
(\bigwedge a\ b\ c\ p. \\
\quad (p,\ a,\ b) \in \mathit{rtranclR}\ r^R \Longrightarrow \\
\quad P\ p\ a\ b \Longrightarrow r^R\ (b,\ c) \Longrightarrow P\ (\mathit{rtranclT.rtrancl\text{-}into\text{-}rtrancl}\ a\ b\ c\ p)\ a\ c) \Longrightarrow \\
P\ p\ x\ y
\end{array}$$

[2]wrt. set inclusion

The proof can be formalized in Isabelle as follows:

theorem *induct-correctness*:
  assumes R: $(p,\ x,\ y) \in \mathit{rtranclR}\ r^R$
    — p is a realizer for $(x,\ y) \in r^*$
  and f: $\bigwedge a.\ P^R\ (f\ a)\ a\ a$
    — f is a realizer for the induction basis
  and g: $\bigwedge a\ b\ c\ p\ q.\ (p,\ a,\ b) \in \mathit{rtranclR}\ r^R \Longrightarrow P^R\ q\ a\ b \Longrightarrow$
    $r^R\ (b,\ c) \Longrightarrow P^R\ (g\ a\ b\ c\ p\ q)\ a\ c$
    — g is a realizer for the induction step
  shows $P^R\ (\mathit{rtranclT\text{-}rec}\ f\ g\ p)\ x\ y$ **using** R
proof *induct* — induction on the derivation of R
  fix a
  show $P^R\ (\mathit{rtranclT\text{-}rec}\ f\ g\ (\mathit{rtranclT.rtrancl\text{-}refl}\ a))\ a\ a$ **by** *simp* (*rule* f)
next
  fix $a\ b\ c\ p$
  assume $(p,\ a,\ b) \in \mathit{rtranclR}\ r^R$ **and** $P^R\ (\mathit{rtranclT\text{-}rec}\ f\ g\ p)\ a\ b$ **and** $r^R\ (b,\ c)$
  show $P^R\ (\mathit{rtranclT\text{-}rec}\ f\ g\ (\mathit{rtranclT.rtrancl\text{-}into\text{-}rtrancl}\ a\ b\ c\ p))\ a\ c$
    by *simp* (*rule* g)
qed

As in the case of inductive datatypes, there is also a weaker *case analysis* rule

$$z \in r^* \Longrightarrow$$
$$(\bigwedge a.\ z = (a,\ a) \Longrightarrow P) \Longrightarrow (\bigwedge a\ b\ c.\ z = (a,\ c) \Longrightarrow (a,\ b) \in r^* \Longrightarrow (b,\ c) \in r \Longrightarrow P) \Longrightarrow P$$

for the predicate *rtrancl*, which is realized by the case analysis combinator

$$\mathit{rtranclT\text{-}case}\ f\ g\ (\mathit{rtranclT.rtrancl\text{-}refl}\ a) = f\ a$$
$$\mathit{rtranclT\text{-}case}\ f\ g\ (\mathit{rtranclT.rtrancl\text{-}into\text{-}rtrancl}\ a\ b\ c\ p) = g\ a\ b\ c\ p$$

for the datatype *rtranclT*. The correctness of this realizer is expressed by the rule

$$(p,\ z) \in \mathit{rtranclR}\ r^R \Longrightarrow$$
$$(\bigwedge a.\ z = (a,\ a) \Longrightarrow P^R\ (f\ a)) \Longrightarrow$$
$$(\bigwedge a\ b\ c.\ z = (a,\ c) \Longrightarrow (\bigwedge x.\ (x,\ a,\ b) \in \mathit{rtranclR}\ r^R \Longrightarrow r^R\ (b,\ c) \Longrightarrow P^R\ (g\ a\ b\ c\ x))) \Longrightarrow$$
$$P^R\ (\mathit{rtranclT\text{-}case}\ f\ g\ p)$$

It can be proved by case analysis on the derivation of $(p,\ z) \in \mathit{rtranclR}\ r^R$, which is expressed by the rule[3]

$$z \in \mathit{rtranclR}\ r^R \Longrightarrow$$
$$(\bigwedge a.\ z = (\mathit{rtranclT.rtrancl\text{-}refl}\ a,\ a,\ a) \Longrightarrow P) \Longrightarrow$$
$$(\bigwedge a\ b\ c\ p.$$
$$\quad z = (\mathit{rtranclT.rtrancl\text{-}into\text{-}rtrancl}\ a\ b\ c\ p,\ a,\ c) \Longrightarrow$$
$$\quad (p,\ a,\ b) \in \mathit{rtranclR}\ r^R \Longrightarrow r^R\ (b,\ c) \Longrightarrow P) \Longrightarrow$$
$$P$$

The correctness theorem can be proved in Isabelle as follows:

[3]Note that $(a,\ b,\ c)$ just abbreviates $(a,\ (b,\ c))$

lemma *elim-correctness*:
  assumes R: $(p,\ z) \in$ *rtranclR* r^R
    — p is a realizer for $z \in r^*$
  and f: $\bigwedge a.\ z = (a,\ a) \Longrightarrow P^R\ (f\ a)$
    — f is a realizer for the base case
  and g: $\bigwedge a\ b\ c\ q.\ z = (a,\ c) \Longrightarrow (q,\ a,\ b) \in \textit{rtranclR}\ r^R \Longrightarrow r^R\ (b,\ c) \Longrightarrow P^R\ (g\ a\ b\ c\ q)$
    — g is a realizer for the step case
  shows P^R (*rtranclT-case* $f\ g\ p$) **using** R
proof *cases* — case analysis on the derivation of R
  fix a
  assume $(p,\ z) = (\textit{rtranclT.rtrancl-refl}\ a,\ (a,\ a))$
  thus P^R (*rtranclT-case* $f\ g\ p$) **by** *simp* (*rule* f, *rules*)
next
  fix $a\ b\ c\ q$
  assume $(p,\ z) = (\textit{rtranclT.rtrancl-into-rtrancl}\ a\ b\ c\ q,\ (a,\ c))$
  and $(q,\ a,\ b) \in$ *rtranclR* r^R **and** $r^R\ (b,\ c)$
  thus P^R (*rtranclT-case* $f\ g\ p$) **by** *simp* (*rule* g, *rules*)
qed

4.3.5.2 General scheme

We now come to a general treatment of realizability for inductive predicates. Consider the general case of an inductive definition

$$\begin{array}{l}
\textsf{inductive}\ S_1\ \overline{a}\ \ldots\ S_n\ \overline{a} \\
\quad I_1^1 : \bigwedge \overline{x_1^1}.\ \varphi_{1,1}^1 \Longrightarrow \cdots \Longrightarrow \varphi_{1,m_1^1}^1 \Longrightarrow \left(\overline{t_1^1}\right) \in S_1\ \overline{a} \\
\quad\quad \vdots \\
\quad I_{k_1}^1 : \bigwedge \overline{x_{k_1}^1}.\ \varphi_{k_1,1}^1 \Longrightarrow \cdots \Longrightarrow \varphi_{k_1,m_{k_1}^1}^1 \Longrightarrow \left(\overline{t_{k_1}^1}\right) \in S_1\ \overline{a} \\
\quad\quad \vdots \\
\quad I_1^n : \bigwedge \overline{x_1^n}.\ \varphi_{1,1}^n \Longrightarrow \cdots \Longrightarrow \varphi_{1,m_1^n}^n \Longrightarrow \left(\overline{t_1^n}\right) \in S_n\ \overline{a} \\
\quad\quad \vdots \\
\quad I_{k_n}^n : \bigwedge \overline{x_{k_n}^n}.\ \varphi_{k_n,1}^n \Longrightarrow \cdots \Longrightarrow \varphi_{k_n,m_{k_n}^n}^n \Longrightarrow \left(\overline{t_{k_n}^n}\right) \in S_n\ \overline{a}
\end{array}$$

All inductive predicates have a common list of *parameters* $\overline{a}$ which remain fixed throughout the inductive definition. In an expression of the form $(\overline{x}) \in S_j\ \overline{a}$, we call $\overline{a}$ the *fixed arguments* and $\overline{x}$ the *flexible arguments* of predicate S_j. By analogy to the case of inductive datatypes, we call a premise $\varphi_{i,i'}^j$ of an introduction rule *recursive*, if it contains any of the newly defined inductive predicates $S_1, \ldots, S_n$, otherwise *nonrecursive*. We denote by $r_{i,1}^j, \ldots, r_{i,l_i^j}^j$ the positions of the recursive premises of the i-th introduction rule of the j-th predicate. A recursive premise $\varphi_{i,r_{i,l}^j}^j$ must have the form

$$\bigwedge \overline{z_{i,l}^j}.\ \overline{\psi_{i,l}^j} \Longrightarrow \left(\overline{u_{i,l}^j}\right) \in S_{s_{i,l}^j}\ \overline{a}$$

where $1 \leq l \leq l_i^j$ and $\overline{\psi_{i,l}^j}$ does *not* contain any of the newly defined inductive predicates. This means that $S_1, \ldots, S_n$ may only occur *strictly positive* in $\varphi_{i,i'}^j$. It should be noted that the general theory of inductive definitions in HOL even admits inductive predicates which are just *weakly positive*. For example, a predicate S with an introduction rule of the form

$$((t \in S \Longrightarrow \cdots) \Longrightarrow \cdots) \Longrightarrow u \in S$$

would also be legal in HOL. However, it turns out that weakly positive inductive definitions are unsuitable for the purpose of program extraction: Since weakly positive recursive occurrences of types are not allowed in HOL datatype definitions [21, §4.2], it will be impossible to define a datatype representing a derivation of S in HOL.

Induction The rule for simultaneous induction on the derivations of S_1, …, S_n has the form

$$\begin{array}{l}\mathcal{I}_1^1 \Longrightarrow \cdots \Longrightarrow \mathcal{I}_{k_1}^1 \Longrightarrow \cdots \Longrightarrow \mathcal{I}_1^n \Longrightarrow \cdots \Longrightarrow \mathcal{I}_{k_n}^n \Longrightarrow \\ \qquad (\overline{x_1} \in S_1\ \overline{a} \longrightarrow P_1\ \overline{x_1}) \wedge \ldots \wedge (\overline{x_n} \in S_n\ \overline{a} \longrightarrow P_n\ \overline{x_n})\end{array}$$

where

$$\begin{array}{l}\mathcal{I}_i^j = \bigwedge \overline{x_i^j}.\ \varphi_{i,1}^j \Longrightarrow \cdots \Longrightarrow \varphi_{i,m_i^j}^j \Longrightarrow \\ \qquad \left(\bigwedge \overline{z_{i,1}^j}.\ \overline{\psi_{i,1}^j} \Longrightarrow P_{s_{i,1}^j}\ \overline{u_{i,1}^j}\right) \Longrightarrow \cdots \Longrightarrow \left(\bigwedge \overline{z_{i,l_i^j}^j}.\ \overline{\psi_{i,l_i^j}^j} \Longrightarrow P_{s_{i,l_i^j}^j}\ \overline{u_{i,l_i^j}^j}\right) \Longrightarrow P_j\ \overline{t_i^j}\end{array}$$

Case analysis The rule for case analysis on the derivation of S_j has the form

$$(\overline{z}) \in S_j\ \overline{a} \Longrightarrow \mathcal{I}_1^j \Longrightarrow \cdots \Longrightarrow \mathcal{I}_{k_j}^j \Longrightarrow P$$

where

$$\mathcal{I}_i^j\ \overline{z} = \bigwedge \overline{x_i^j}.\ (\overline{z}) = \left(\overline{t_i^j}\right) \Longrightarrow \varphi_{i,1}^j \Longrightarrow \cdots \Longrightarrow \varphi_{i,m_i^j}^j \Longrightarrow P$$

Computational content of derivations To simplify the presentation, assume that all premises of the introduction rules have a computational content. Then the datatype representing the computational content of the derivations of $\overline{x_1} \in S_1\ \overline{a}$, …, $\overline{x_n} \in S_n\ \overline{a}$ is

$$\begin{array}{l}\mathsf{datatype}\ (\overline{\alpha_P}, \overline{\alpha})\ S_1^T\ = \\ \quad I_1^1\ \overline{\tau_1^1}\ \left(\mathsf{typeof}\ \varphi_{1,1}^1\right)\ \ldots\ \left(\mathsf{typeof}\ \varphi_{1,m_1^1}^1\right)\ \mid\ \ldots\ \mid\ I_{k_1}^1\ \overline{\tau_{k_1}^1}\ \left(\mathsf{typeof}\ \varphi_{k_1,1}^1\right)\ \ldots\ \left(\mathsf{typeof}\ \varphi_{k_1,m_{k_1}^1}^1\right) \\ \qquad\qquad \vdots \\ \mathsf{and}\ (\overline{\alpha_P}, \overline{\alpha})\ S_n^T\ = \\ \quad I_1^n\ \overline{\tau_1^n}\ \left(\mathsf{typeof}\ \varphi_{1,1}^n\right)\ \ldots\ \left(\mathsf{typeof}\ \varphi_{1,m_1^n}^n\right)\ \mid\ \ldots\ \mid\ I_{k_n}^n\ \overline{\tau_{k_n}^n}\ \left(\mathsf{typeof}\ \varphi_{k_n,1}^n\right)\ \ldots\ \left(\mathsf{typeof}\ \varphi_{k_n,m_{k_n}^n}^n\right)\end{array}$$

Intuitively, the list of argument types of a constructor I_i^j corresponding to an introduction rule consists of the types of the variables $\overline{x_i^j}$ occurring in the introduction rule, and the types extracted from the premises of the rule[4]. In the above definition, $\overline{\alpha}$ is the list of type variables occurring in the introduction rules $I_{i,i'}^j$ and $\overline{\alpha_P}$ is the list of all type variables representing the computational content of the computationally relevant predicate variables $\overline{P}$ occurring in $\overline{a}$, i.e. $\overline{P} \subseteq \overline{a}$. For the rest of this section, we will assume that all predicate variables occurring in $\overline{a}$ are computationally relevant. The set of equations characterizing the function typeof is augmented with the equations

$$\mathsf{typeof}\ \overline{P} \equiv \overline{\alpha_P} \Longrightarrow \mathsf{typeof}\ ((\overline{x_j}) \in S_j\ \overline{a}) \equiv (\overline{\alpha_P}, \overline{\alpha})\ S_j^T$$

For reasons which have already been discussed in §4.2.1, it may in general be necessary to generate up to 2^n variants of the datatypes S_j^T, where n is the number of predicate variables occurring in $\overline{a}$.

[4]To guarantee the nonemptiness of the datatypes S_1^T, …, S_n^T, which is a fundamental requirement for any HOL type, it may be necessary to add extra *dummy constructors* to the above definition. See §4.3.5.3 for an example.

Realizability predicate The realizability predicates S_1^R, …, S_n^R, which establish a connection between elements of the datatypes S_1^T, …, S_n^T and propositions of the form $(\overline{x_1}) \in S_1\ \overline{a}$, …, $(\overline{x_n}) \in S_n\ \overline{a}$, are defined inductively using the introduction rules

$$\bigwedge \overline{x_i^j}\ p_1.\ \mathsf{realizes}\ p_1\ \varphi_{i,1}^j \Longrightarrow \left(\cdots \Longrightarrow \left(\bigwedge p_{m_i^j}.\ \mathsf{realizes}\ p_{m_i^j}\ \varphi_{i,m_i^j}^j \Longrightarrow \left(I_i^j\ \overline{x_i^j}\ \overline{p_i^j}, \overline{t_i^j}\right) \in S_j^R\ \overline{a^R}\right)\cdots\right)$$

where $1 \leq j \leq n$ and $1 \leq i \leq k_j$. The set of equations characterizing the function realizes is augmented with the equations

$$\begin{aligned}&\mathsf{typeof}\ \overline{P} \equiv \overline{\alpha_P} \Longrightarrow \mathsf{realizes}\ r\ ((\overline{x_j}) \in S_j\ \overline{a})\ \equiv \\ &\quad (r,\ \overline{x_j}) \in S_j^R\ (\lambda p\ \overline{z_1}.\ \mathsf{realizes}\ p\ (P_1\ \overline{z_1}))\ \cdots\ (\lambda p\ \overline{z_n}.\ \mathsf{realizes}\ p\ (P_n\ \overline{z_n}))\ (\overline{a}\backslash\overline{P})\end{aligned}$$

As before, $\overline{P} = \{P_1, \ldots, P_n\} \subseteq \overline{a}$, denotes the list of fixed arguments which are predicates. To simplify the notation, we assume that the argument list of S_j is sorted such that predicate arguments come first. Predicate arguments of S_j are turned into suitable realizability predicates on the right-hand side of the above equation, whereas non-predicate arguments are left unchanged.

Realizer for induction rule To simplify the presentation, we again assume that all of the predicates P_1, …, P_n have a computational content. Then, the rule for simultaneous induction on the derivations of S_1, …, S_n presented above is realized by the term

$$\lambda f_1^1 \ldots f_{k_n}^n.\ \left(S_1^T\text{-}rec\ f_1^1 \ldots f_{k_n}^n,\ \ \ldots,\ S_n^T\text{-}rec\ f_1^1 \ldots f_{k_n}^n\right)$$

The fact that this term is a correct realizer can be expressed as follows

$$\begin{aligned}&\mathcal{R}_1^1\ f_1^1 \Longrightarrow \cdots \Longrightarrow \mathcal{R}_{k_1}^1\ f_{k_1}^1 \Longrightarrow \cdots \Longrightarrow \mathcal{R}_1^n\ f_1^n \Longrightarrow \cdots \Longrightarrow \mathcal{R}_{k_n}^n\ f_{k_n}^n \Longrightarrow \\ &\quad \Big(\forall q.\ (q, \overline{x_1}) \in S_1^R\ \overline{a^R} \longrightarrow \\ &\qquad P_1^R\ \left(fst\ \left(\cdots\left(fst\ \left(S_1^T\text{-}rec\ f_1^1 \ldots f_{k_n}^n,\ \ \ldots,\ S_n^T\text{-}rec\ f_1^1 \ldots f_{k_n}^n\right)\right)\cdots\right)\ q\right)\ \overline{x_1}\Big) \wedge \ldots \wedge \\ &\quad \Big(\forall q.\ (q, \overline{x_n}) \in S_n^R\ \overline{a^R} \longrightarrow \\ &\qquad P_n^R\ \left(snd\ \left(\cdots\left(snd\ \left(S_1^T\text{-}rec\ f_1^1 \ldots f_{k_n}^n,\ \ \ldots,\ S_n^T\text{-}rec\ f_1^1 \ldots f_{k_n}^n\right)\right)\cdots\right)\ q\right)\ \overline{x_n}\Big)\end{aligned}$$

where

$$\begin{aligned}&\mathcal{R}_i^j\ f = \bigwedge \overline{x_i^j}\ p_1.\ \mathsf{realizes}\ p_1\ \varphi_{i,1}^j \Longrightarrow \Big(\cdots \Longrightarrow \Big(\bigwedge p_{m_i^j}.\ \mathsf{realizes}\ p_{m_i^j}\ \varphi_{i,m_i^j}^j \Longrightarrow \\ &\quad \left(\bigwedge q_1.\ \left(\bigwedge \overline{z_{i,1}^j}\ \overline{y_{i,1}^j}.\ \mathsf{realizes}\ \overline{y_{i,1}^j}\ \overline{\psi_{i,1}^j} \Longrightarrow P_{s_{i,1}^j}^R\ \left(q_1\ \overline{z_{i,1}^j}\ \overline{y_{i,1}^j}\right)\ \overline{u_{i,1}^j}\right) \Longrightarrow \Big(\cdots \Longrightarrow \right. \\ &\quad \left(\bigwedge q_{l_i^j}.\ \left(\bigwedge \overline{z_{i,l_i^j}^j}\ \overline{y_{i,l_i^j}^j}.\ \mathsf{realizes}\ \overline{y_{i,l_i^j}^j}\ \overline{\psi_{i,l_i^j}^j} \Longrightarrow P_{s_{i,l_i^j}^j}^R\ \left(q_{l_i^j}\ \overline{z_{i,l_i^j}^j}\ \overline{y_{i,l_i^j}^j}\right)\ \overline{u_{i,l_i^j}^j}\right) \Longrightarrow \right. \\ &\qquad P_j^R\ \left(f\ \overline{x_i^j}\ \overline{p_i^j}\ \overline{q_i^j}\right)\ \overline{t_i^j}\Big)\cdots\Big)\Big)\Big)\cdots\Big) \\ &\overline{p_i^j} = p_1 \ldots p_{m_i^j} \quad \text{and} \quad \overline{q_i^j} = q_1 \ldots q_{l_i^j}\end{aligned}$$

This correctness theorem is proved by simultaneous induction on the derivations of $S_1^R \ldots S_n^R$. Assume we have $\mathcal{R}_1^1\ f_1^1$, …, $\mathcal{R}_{k_n}^n\ f_{k_n}^n$. For introduction rule I_i^j, we obtain the induction hypotheses

$$\begin{aligned}&\bigwedge \overline{z_{i,1}^j}\ \overline{y_{i,1}^j}.\ \mathsf{realizes}\ \overline{y_{i,1}^j}\ \overline{\psi_{i,1}^j} \Longrightarrow P_{s_{i,1}^j}^R\ \left(S_{s_{i,1}^j}^T\text{-}rec\ f_1^1 \ldots f_{k_n}^n\ \left(p_{r_{i,1}^j}\ \overline{z_{i,1}^j}\ \overline{y_{i,1}^j}\right)\right)\ \overline{u_{i,1}^j} \\ &\qquad\qquad \vdots \\ &\bigwedge \overline{z_{i,l_i^j}^j}\ \overline{y_{i,l_i^j}^j}.\ \mathsf{realizes}\ \overline{y_{i,l_i^j}^j}\ \overline{\psi_{i,l_i^j}^j} \Longrightarrow P_{s_{i,l_i^j}^j}^R\ \left(S_{s_{i,l_i^j}^j}^T\text{-}rec\ f_1^1 \ldots f_{k_n}^n\ \left(p_{r_{i,l_i^j}^j}\ \overline{z_{i,l_i^j}^j}\ \overline{y_{i,l_i^j}^j}\right)\right)\ \overline{u_{i,l_i^j}^j}\end{aligned}$$

as well as

$$\text{realizes}\ p_1\ \varphi^j_{i,1} \qquad \ldots \qquad \text{realizes}\ p_{m^j_i}\ \varphi^j_{i,m^j_i}$$

from which we have to show

$$P^R_j\ \left(S^T_j\textit{-rec}\ f^1_1 \ldots f^n_{k_n}\ \left(I^j_i\ \overline{x^j_i}\ \overline{p^j_i}\right)\right)\ \overline{t^j_i}$$

for all $\overline{x^j_i}$ and $\overline{p^j_i}$. This is equivalent to

$$P^R_j\ \left(f^j_i\ \overline{x^j_i}\ \overline{p^j_i}\ q_1 \ldots q_{l^j_i}\right)\ \overline{t^j_i}$$

where

$$q_l = \lambda \overline{z^j_{i,l}}\ \overline{y^j_{i,l}}.\ S^T_{s^j_{i,l}}\textit{-rec}\ f^1_1 \ldots f^n_{k_n}\ \left(p_{r^j_{i,l}}\ \overline{z^j_{i,l}}\ \overline{y^j_{i,l}}\right)$$

which can easily be deduced from the induction hypotheses using $\mathcal{R}^j_i\ f^j_i$.

Realizer for case analysis rule The rule for case analysis on the derivation of S_j is realized by the case analysis combinator for the datatype S^T_j, i.e. by the term

$$\lambda q\ f_1 \ldots f_{k_j}.\ S^T_j\textit{-case}\ f_1 \ldots f_{k_j}\ q$$

The correctness of this realizer is expressed by the formula

$$(q, \overline{z}) \in S^R_j\ \overline{a^R} \Longrightarrow \mathcal{R}^j_1\ f_1\ \overline{z} \Longrightarrow \cdots \Longrightarrow \mathcal{R}^j_{k_j}\ f_{k_j}\ \overline{z} \Longrightarrow P^R\ (S^T_j\textit{-case}\ f_1 \ldots f_{k_j}\ q)$$

where

$$\begin{array}{l}\mathcal{R}^j_i\ f\ \overline{z} = \bigwedge \overline{x^j_i}.\ (\overline{z}) = (\overline{t^j_i}) \Longrightarrow \\ \quad \left(\bigwedge p_1.\ \text{realizes}\ p_1\ \varphi^j_{i,1} \Longrightarrow \left(\cdots \Longrightarrow \left(\bigwedge p_{m^j_i}.\ \text{realizes}\ p_{m^j_i}\ \varphi^j_{i,m^j_i} \Longrightarrow P^R\ \left(f\ \overline{x^j_i}\ \overline{p^j_i}\right)\right)\right)\right)\end{array}$$

This correctness theorem can be proved by case analysis on the derivation of $(q, \overline{z}) \in S^R_j\ \overline{a^R}$. Assume we have $\mathcal{R}^j_1\ f_1\ \overline{z}$, ..., $\mathcal{R}^j_{k_j}\ f_{k_j}\ \overline{z}$. For introduction rule I^j_i, we get the assumptions

$$(q,\ \overline{z}) = \left(I^j_i\ \overline{x^j_i}\ \overline{p^j_i},\ \overline{t^j_i}\right) \quad \text{realizes}\ p_1\ \varphi^j_{i,1} \qquad \ldots \qquad \text{realizes}\ p_{m^j_i}\ \varphi^j_{i,m^j_i}$$

from which we have to show

$$P^R\ (S^T_j\textit{-case}\ f_1 \ldots f_{k_j}\ q)$$

for all $\overline{x^j_i}$ and $\overline{p^j_i}$. Due to the above assumption on the structure of q, this is equivalent to

$$P^R\ \left(f_j\ \overline{x^j_i}\ \overline{p^j_i}\right)$$

which easily follows from the above assumptions using $\mathcal{R}^j_i\ f_i\ \overline{z}$.

Inductive predicates without computational content As is the case for other formulae, there may also be inductive predicates that are only intended for specification purposes, but should not contribute to the computational content of a proof. Our framework relies on a *declarative* approach here, which means that the author of a formalization has to *specify* whether or not an inductive predicate should have a computational content.

A particular restriction, which has to be taken care of in this context is that a proof must not contain any proofs of computationally *relevant* statements by induction or case analysis on the derivation of an inductive predicate, which has been declared to be computationally *irrelevant* by the user. In this case, there is no way to construct a program from such a proof, since a datatype representing the derivation over which to do recursion in the program is not available, and hence program extraction fails. As will be described in §5.1.2, similar restrictions on proofs by induction are also imposed by the Coq system [12, §4.5.4].

Moreover, as has already been noted in §4.3.2, we do not in general have $\mathsf{realizes}\ \mathsf{Null}\ ((\overline{y}) \in S) = (\overline{y}) \in S$ for an arbitrary inductive predicate S with $\mathsf{typeof}\ ((\overline{y}) \in S) = \mathsf{Null}$. In particular, this equation does not hold for inductive predicates *without* computational content whose introduction rules $I : \bigwedge \overline{x}.\ \overline{\varphi} \Longrightarrow (\overline{t}) \in S$ have premises containing inductive predicates *with* computational content, and hence we do not have

$$\mathsf{realizes}\ \mathsf{Null}\ (\bigwedge \overline{x}.\ \overline{\varphi} \Longrightarrow (\overline{t}) \in S) = (\bigwedge \overline{x}.\ \overline{\varphi} \Longrightarrow (\overline{t}) \in S)$$

In this case, we need to inductively define a realizability predicate S^R in a similar way as in the case of a computationally relevant predicate, with the difference that S^R has the same number of arguments as S, since there is no argument corresponding to the realizer.

4.3.5.3 Examples

To convince ourselves of the generality of the approach we have just described, it is instructive to have a look at some more examples. As a first example, we will examine inductive characterizations of the usual basic logical operators such as conjunction, disjunction, existential quantification and equality. This inductive characterization, which is due to Paulin-Mohring [87], was first used in the *Coq* theorem prover based on the *Calculus of Inductive Constructions*. As expected, it will turn out that the realizers and realizability predicates generated by Isabelle's program extraction module, which implements the theory described in the previous section, are equivalent to those introduced in §4.3.3.

The inductive predicates *And*, *Or* and *Exists*, which represent conjunction, disjunction and existential quantification, respectively, have predicate variables P and Q as their only *fixed* arguments. Note that these predicates have no *flexible* arguments. Thus, the conlusions of the corresponding introduction rules have the somewhat degenerate form $() \in \ldots$, where the *unit element* $()$ denotes the empty argument tuple. Intuitively, we can also think of these predicates to denote the empty set, if the proposition is *False*, and the singleton set consisting of just the element $()$, if the proposition is *True*. In the sequel, we will assume that both P and Q are computationally relevant.

The *And* predicate has just one introduction rule:

consts $And :: bool \Rightarrow bool \Rightarrow unit\ set$
inductive $And\ P\ Q$
intros $AndI$: $P \Longrightarrow Q \Longrightarrow () \in And\ P\ Q$

The introduction rule *AndI* is realized by a constructor with two arguments. The corresponding realizability predicate is characterized by the introduction rule

$P^R\ p \Longrightarrow Q^R\ q \Longrightarrow (\mathit{AndT\text{-}P\text{-}Q.AndI}\ p\ q,\ ()) \in \mathit{AndR\text{-}P\text{-}Q}\ P^R\ Q^R$

The induction rule for predicate *And*

$u \in \mathit{And}\ P\ Q \Longrightarrow (P \Longrightarrow Q \Longrightarrow R\ ()) \Longrightarrow R\ u$

is the usual rule for conjunction elimination. Here, u can only be the unit element. This elimination rule is realized by[5]

$\mathit{AndT\text{-}P\text{-}Q\text{-}rec}\ f\ (\mathit{AndT\text{-}P\text{-}Q.AndI}\ P\ Q) = f\ P\ Q$

The fact that this is a correct realizer is expressed by

$(r,\ u) \in \mathit{AndR\text{-}P\text{-}Q}\ P^R\ Q^R \Longrightarrow$
$(\bigwedge x.\ P^R\ x \Longrightarrow (\bigwedge xa.\ Q^R\ xa \Longrightarrow R^R\ (f\ x\ xa)\ ())) \Longrightarrow R^R\ (\mathit{AndT\text{-}P\text{-}Q\text{-}rec}\ f\ r)\ u$

Analogously, the *Or* predicate is characterized by two introduction rules:

consts $\mathit{Or} :: \mathit{bool} \Rightarrow \mathit{bool} \Rightarrow \mathit{unit\ set}$
inductive $\mathit{Or}\ P\ Q$
intros
  $\mathit{OrI1}$: $P \Longrightarrow () \in \mathit{Or}\ P\ Q$
  $\mathit{OrI2}$: $Q \Longrightarrow () \in \mathit{Or}\ P\ Q$

The introduction rules *OrI1* and *OrI2* are realized by constructors, each of which has one argument representing the computational content of P and Q, respectively. The corresponding realizability predicate is characterized by the introduction rules

$P^R\ p \Longrightarrow (\mathit{OrT\text{-}P\text{-}Q.OrI1}\ p,\ ()) \in \mathit{OrR\text{-}P\text{-}Q}\ P^R\ Q^R$
$Q^R\ q \Longrightarrow (\mathit{OrT\text{-}P\text{-}Q.OrI2}\ q,\ ()) \in \mathit{OrR\text{-}P\text{-}Q}\ P^R\ Q^R$

The induction rule for predicate *Or*

$u \in \mathit{Or}\ P\ Q \Longrightarrow (P \Longrightarrow R\ ()) \Longrightarrow (Q \Longrightarrow R\ ()) \Longrightarrow R\ u$

is the usual rule for disjunction elimination. This elimination rule is realized by

$\mathit{OrT\text{-}P\text{-}Q\text{-}rec}\ f\ g\ (\mathit{OrT\text{-}P\text{-}Q.OrI1}\ P) = f\ P$
$\mathit{OrT\text{-}P\text{-}Q\text{-}rec}\ f\ g\ (\mathit{OrT\text{-}P\text{-}Q.OrI2}\ Q) = g\ Q$

The correctness theorem for this realizer is

$(r,\ u) \in \mathit{OrR\text{-}P\text{-}Q}\ P^R\ Q^R \Longrightarrow$
$(\bigwedge x.\ P^R\ x \Longrightarrow R^R\ (f\ x)\ ()) \Longrightarrow$
$(\bigwedge x.\ Q^R\ x \Longrightarrow R^R\ (g\ x)\ ()) \Longrightarrow R^R\ (\mathit{OrT\text{-}P\text{-}Q\text{-}rec}\ f\ g\ r)\ u$

The *Exists* predicate can be characterized inductively as follows:

consts $\mathit{Exists} :: ('a \Rightarrow \mathit{bool}) \Rightarrow \mathit{unit\ set}$
inductive $\mathit{Exists}\ P$
intros $\mathit{ExistsI}$: $P\ x \Longrightarrow () \in \mathit{Exists}\ P$

The introduction rule *ExistsI* is realized by a constructor with two arguments: One argument is the value of variable x, whereas the other is the computational content of $P\ x$. The corresponding realizability predicate is characterized by the introduction rule

[5]Note that the P and Q in *AndT-P-Q-rec* are *not* parameters, but are part of the function name. Since a fixed parameter of an inductive predicate may either be computationally relevant or irrelevant, there may be several variants of this function, which are distinguished by suffixing their names with the names of the computationally relevant parameters.

$P^R\ p\ x \Longrightarrow (\mathit{ExistsT\text{-}P.ExistsI}\ x\ p,\ ()) \in \mathit{ExistsR\text{-}P}\ P^R$

The induction rule for predicate *Exists*

$u \in \mathit{Exists}\ P \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ ()) \Longrightarrow Q\ u$

is the usual rule for existential elimination. This elimination rule is realized by

$\mathit{ExistsT\text{-}P\text{-}rec}\ f\ \ (\mathit{ExistsT\text{-}P.ExistsI}\ a\ p) = f\ a\ p$

The correctness theorem for this realizer is

$(r,\ u) \in \mathit{ExistsR\text{-}P}\ P^R \Longrightarrow$
$(\bigwedge x\ xa.\ P^R\ xa\ x \Longrightarrow Q^R\ (f\ x\ xa)\ ()) \Longrightarrow Q^R\ (\mathit{ExistsT\text{-}P\text{-}rec}\ f\ r)\ u$

Finally, we come to the inductive characterization of equality. The equality predicate *Eq* has one fixed and one flexible argument, which are both of the same type. The formula $x \in Eq$ y should be read as "x is equal to y". Intuitively, $Eq\ a$ can be thought of as the set of all elements which are equal to a, which is the singleton set consisting of just the element a.

consts $Eq :: {'a} \Rightarrow {'a}\ set$
inductive $Eq\ a$
intros *Refl*: $a \in Eq\ a$

The only introduction rule for equality is reflexivity. Since fixed arguments are not part of the datatype representing the computational content of *Eq* and the flexible argument of *Eq* in rule *Refl* coincides with the fixed argument, the realizer for rule *Refl* is just a constructor with no arguments. Thus, the realizability predicate for *Eq* is characterized by the rather trivial rule

$(\mathit{Refl},\ a) \in \mathit{EqR}\ a$

The induction rule for predicate *Eq*

$x \in Eq\ y \Longrightarrow P\ y \Longrightarrow P\ x$

corresponds to the well-known substitution rule. This rule is realized by

$\mathit{EqT\text{-}rec}\ f\ \mathit{Refl} = f$

This is essentially the identity function, which confirms our observation made in §4.3.3 concerning the computational content of the substitution rule. The correctness of this realizer is expressed by

$(e,\ x) \in \mathit{EqR}\ y \Longrightarrow P^R\ p\ y \Longrightarrow P^R\ (\mathit{EqT\text{-}rec}\ p\ e)\ x$

It should be noted that the constructor *Refl* is actually useless, since it does not convey any information which contributes to the computation of *EqT-rec*.

As a more advanced example, we consider the *accessible part* of a relation r. It is characterized by the introduction rule

accI: $(\bigwedge y.\ (y,\ x) \in r \Longrightarrow y \in acc\ r) \Longrightarrow x \in acc\ r$

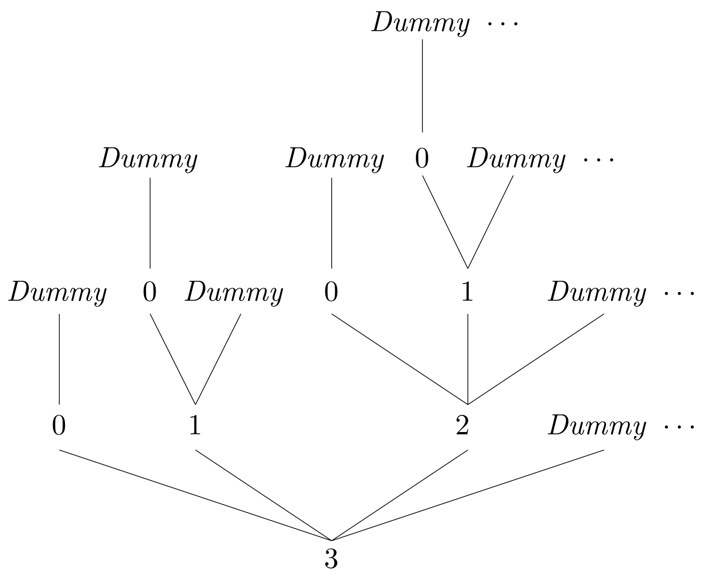


Figure 4.3: Realizer for $\mathit{3} \in \mathit{acc}\ \{(x,y).\ x < y\}$

For the case where r is computationally irrelevant, the computational content of a derivation of $x \in \mathit{acc}\ r$ can be represented by the infinitely branching datatype

$$\textsf{datatype}\ \alpha\ \mathit{accT}\ =\ \mathit{Dummy}\ \mathit{unit}\ \mid\ \mathit{accI}\ \alpha\ (\alpha \Rightarrow \alpha\ \mathit{accT})$$

Note that this datatype has an additional *Dummy* constructor, because it would otherwise be empty. The corresponding realizability predicate is characterized by the rule

$$(\bigwedge y.\ r\ (y,\ x) \Longrightarrow (g\ y,\ y) \in \mathit{accR}\ r) \Longrightarrow (\mathit{accT.accI}\ x\ g,\ x) \in \mathit{accR}\ r$$

Note that there is no rule for constructor *Dummy*, since it does not constitute a proper realizer. Figure 4.3 shows a realizer for the accessibility of *3* with respect to the relation $<$ on natural numbers. The rule for induction on the derivation of *acc* is

$$\begin{array}{l} x \in \mathit{acc}\ r \Longrightarrow \\ (\bigwedge x.\ (\bigwedge y.\ (y,\ x) \in r \Longrightarrow y \in \mathit{acc}\ r) \Longrightarrow (\bigwedge y.\ (y,\ x) \in r \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow P\ x \end{array}$$

This rule is realized by the recursion combinator for *accT*

$$\begin{array}{l} \mathit{accT\text{-}rec}\ \mathit{err}\ f\ \ (\mathit{accT.Dummy}\ u) = \mathit{err}\ u \\ \mathit{accT\text{-}rec}\ \mathit{err}\ f\ \ (\mathit{accT.accI}\ x\ g) = f\ x\ g\ \ (\lambda y.\ \mathit{accT\text{-}rec}\ \mathit{err}\ f\ \ (g\ y)) \end{array}$$

It is interesting to note that this function actually allows to simulate *well-founded recursion* by *primitive recursion* on a datatype encoding the termination relation. Recursive calls to $\lambda y.\ \mathit{accT\text{-}rec}\ \mathit{err}\ f\ \ (g\ y)$ are only guaranteed to yield meaningful results for arguments y with $(y,\ x) \in r$, whereas arguments not satisfying this restriction may cause the function *err* to be called. This behaviour of *accT-rec* is expressed by the correctness theorem

$$\begin{array}{l} (a,\ x) \in \mathit{accR}\ r^R \Longrightarrow \\ (\bigwedge x\ g.\ (\bigwedge y.\ r^R\ (y,\ x) \Longrightarrow (g\ y,\ y) \in \mathit{accR}\ r^R) \Longrightarrow \\ \qquad\qquad (\bigwedge h.\ (\bigwedge y.\ r^R\ (y,\ x) \Longrightarrow P^R\ (h\ y)\ y) \Longrightarrow P^R\ (f\ x\ g\ h)\ x)) \Longrightarrow \\ P^R\ (\mathit{accT\text{-}rec}\ \mathit{arbitrary}\ f\ a)\ x \end{array}$$

When generating executable code, *arbitrary* may be implemented by something like `(fn _ => raise ERROR)`, where the dummy abstraction avoids spurious `ERROR` messages due to ML's eager evaluation.

4.4 Related work

The first theorem provers to support program extraction were Constable's Nuprl system [27], which is based on Martin-Löf type theory, and the PX system by Hayashi [48]. The Coq system [12], which is based on the Calculus of Inductive Constructions (CIC), can extract programs to OCaml [88] and Haskell. Paulin-Mohring [86, 85] has given a realizability interpretation for the Calculus of Constructions and proved the correctness of extracted programs with respect to this realizability interpretation. Although it would be possible in principle to check the correctness proof corresponding to an extracted program inside Coq itself, this has not been implemented yet. Moreover, it is not completely obvious how to do this in practice, because Coq allows for the omission of termination arguments (e.g. wellordering types such as the $accT$ type introduced in §4.3.5.3) in the extracted program, which may render the program untypable in CIC due to the occurrence of unguarded fixpoints [61, §3.4]. Instead of distinguishing between *relevant* and *irrelevant* predicate variables as described in §4.2, the Coq system has two universes Set and Prop, which are inhabited by computationally interesting and computationally noninteresting types, respectively (see also §5.1.2). Recently, Fernández, Severi and Szasz [37, 110] have proposed an extension of the Calculus of Constructions called the *Theory of Specifications*, which internalizes program extraction and realizability. The built-in reduction relation of this calculus reflects the behaviour of the functions corr and extr defined in §4.2. A similar approach is taken in Burstall and McKinna's theory of *deliverables* [67]. A deliverable is a pair consisting of a program together with its correctness proof, which is modeled using strong Σ types. Anderson [4] describes the embedding of a first order logic with program extraction in Elf and proves several meta-theoretic properties of the extraction function, e.g. well-typedness of the extracted program. The Minlog system [15] by Schwichtenberg can extract Scheme programs from proofs in minimal first order logic, enriched with inductive datatypes and predicates. It has recently been extended to produce correctness proofs for extracted programs as well. Moreover, it also supports program extraction from classical proofs [17]. Isabelle has already been used for implementing program extraction calculi in the past, too. Basin and Ayari [7] have shown how to simulate Manna and Waldinger's "Deductive Tableau" in Isabelle/HOL. Coen [26] formalized his own "Classical Computational Logic", which is tailored specifically towards program extraction, whereas our framework is applicable to common object logics such as HOL.

Chapter 5

Case studies

In this chapter, we will present several case studies, demonstrating the practical applicability of the program extraction framework developed in the previous chapter. We start with two relatively simple examples, namely the extraction of an algorithm for computing the quotient and remainder of two natural numbers, as well as Warshall's algorithm for computing the transitive closure of a relation. To show that the extraction mechanism scales up well to larger applications, we then present a formalization of Higman's lemma, as well as a proof of weak normalization for the simply-typed λ-calculus. While the first two examples only involve induction on datatypes, the last two examples also make use of more advanced proof techniques, such as induction on the derivation of inductively defined predicates.

5.1 Quotient and remainder

As an introductory example, we demonstrate how a program for computing the quotient and remainder of two natural numbers can be derived using program extraction. We will also use this example to compare Isabelle's implementation of proof terms and program extraction with the one used in the theorem prover Coq [12].

5.1.1 The Isabelle proof

The specification of the division algorithm is an existential statement of the form

$$\exists r\ q.\ a = \mathit{Suc}\ b * q + r \wedge r \leq b$$

asserting that there exists a remainder r and quotient q for each dividend a and divisor *Suc* b. This formulation avoids an extra precondition stating that the divisor has to be greater than *0*. The proof of this statement is by induction on the dividend a. In the base case, a is *0*, and hence both the quotient and remainder are *0*, too. In the induction step, we need to find a quotient q' and remainder r' for *Suc* a and *Suc* b, given that we already have a quotient and remainder for a and *Suc* b, i.e. r and q such that $a = \mathit{Suc}\ b * q + r$ and $r \leq b$. This is done by considering the cases $r = b$ and $r \neq b$. The case distinction is justified by the lemma

lemma *nat-eq-dec*: $\bigwedge n{::}nat.\ m = n \vee m \neq n$

expressing the decidability of equality on natural numbers, which can be proved constructively by induction on m followed by a case distinction on n. Now if $r = b$, we reset the remainder to

```
theorem division: ∃ r q. a = Suc b * q + r ∧ r ≤ b
proof (induct a)
  case 0
  have 0 = Suc b * 0 + 0 ∧ 0 ≤ b by simp
  thus ?case by rules
next
  case (Suc a)
  then obtain r q where I: a = Suc b * q + r and r ≤ b by rules
  from nat-eq-dec show ?case
  proof
    assume r = b
    with I have Suc a = Suc b * (Suc q) + 0 ∧ 0 ≤ b by simp
    thus ?case by rules
  next
    assume r ≠ b
    hence r < b by (simp add: order-less-le)
    with I have Suc a = Suc b * q + (Suc r) ∧ (Suc r) ≤ b by simp
    thus ?case by rules
  qed
qed
```

Figure 5.1: Proof of existence of quotient and remainder in Isar

0 and increment the quotient, i.e. $r' = 0$ and $q' = \mathit{Suc}\ q$. Otherwise, if $r \neq b$, we also have $r < b$, since $r \leq b$. In this case, we increment the remainder and leave the quotient unchanged, i.e. $r' = \mathit{Suc}\ r$ and $q' = q$. A formalization of this proof in Isabelle/Isar is shown in Figure 5.1. As a result of processing this high-level proof description using the Isabelle/Isar interpreter, a *primitive* proof object is generated, which is shown in Figure 5.2. This proof object is of course much more detailed than its Isar counterpart. While the Isar proof description just abstractly refers to proof methods such as *rules* or *simp*, the corresponding parts of the primitive proof object contain e.g. a sequence of predicate logic rules, such as *exI* or *exE*, or of arithmetic rules for natural numbers, such as *le0* or *Suc-not-Zero*. The shaded subproofs do not contribute to the computational content, but merely verify that the computed results satisfy the specification. They mainly involve lengthy arithmetic reasoning involving numerals, and have therefore been substantially abbreviated for the sake of readability. As it happens, the last of these subproofs even contains classical reasoning by contradiction, using the rule *ccontr*. It is worth noting that this does not affect program extraction, since the formula $\mathit{Suc}\ r \leq b$, which is proved classically, does not have a computational content.

From the primitive proof shown in Figure 5.2, the following program is extracted by Isabelle:

```
division ≡
λa b. nat-rec (0, 0)
        (λn H. case H of
                 (x, y) ⇒ case nat-eq-dec x b of Left ⇒ (0, Suc y) | Right ⇒ (Suc x, y))
        a
```

Since the proof of the *division* theorem relies on the proof of *nat-eq-dec*, extraction of the above program also triggers the extraction of the program

$$
\begin{array}{ll}
\mathit{nat.induct} \cdot (\lambda u.\ \exists\, r\ q.\ u = \mathit{Suc}\ b * q + r \wedge r \le b) \cdot a \bullet & \\
\ (\mathit{exI} \cdot \mathit{TYPE}(\mathit{nat}) \cdot (\lambda x.\ \exists\, q.\ \mathit{0} = \mathit{Suc}\ b * q + x \wedge x \le b) \cdot \mathit{0} \bullet & \longleftarrow \text{remainder} \\
\quad (\mathit{exI} \cdot \mathit{TYPE}(\mathit{nat}) \cdot (\lambda x.\ \mathit{0} = \mathit{Suc}\ b * x + \mathit{0} \wedge \mathit{0} \le b) \cdot \mathit{0} \bullet & \longleftarrow \text{quotient} \\
\qquad (\mathit{conjI} \cdot \mathit{0} = \mathit{Suc}\ b * \mathit{0} + \mathit{0} \cdot \mathit{0} \le b \bullet & \\
\qquad\quad (\mathit{subst} \cdot \mathit{TYPE}(\mathit{nat}) \cdot \mathit{0} \cdot \mathit{Suc}\ b * \mathit{0} + \mathit{0} \cdot (\lambda z.\ \mathit{0} = z) \bullet & \\
\qquad\qquad \cdots \bullet & \\
\qquad\qquad (\mathit{HOL.refl} \cdot \mathit{TYPE}(\mathit{nat}) \cdot \mathit{0})) \bullet & \\
\qquad\quad (\mathit{le0} \cdot b))\ )) \bullet & \\
\ (\boldsymbol{\lambda} n\ H\colon \exists\, x\ xa.\ n = \mathit{Suc}\ b * xa + x \wedge x \le b. & \\
\quad \mathit{exE} \cdot \mathit{TYPE}(\mathit{nat}) \cdot (\lambda r.\ \exists\, x.\ n = \mathit{Suc}\ b * x + r \wedge r \le b) \cdot & \\
\quad \exists\, r\ q.\ \mathit{Suc}\ n = \mathit{Suc}\ b * q + r \wedge r \le b \bullet & \\
\quad H \bullet & \\
\quad (\boldsymbol{\lambda} r\ H\colon \exists\, x.\ n = \mathit{Suc}\ b * x + r \wedge r \le b. & \\
\qquad \mathit{exE} \cdot \mathit{TYPE}(\mathit{nat}) \cdot (\lambda q.\ n = \mathit{Suc}\ b * q + r \wedge r \le b) \cdot & \\
\qquad \exists\, r\ q.\ \mathit{Suc}\ n = \mathit{Suc}\ b * q + r \wedge r \le b \bullet & \\
\qquad H \bullet & \\
\qquad (\boldsymbol{\lambda} q\ H\colon n = \mathit{Suc}\ b * q + r \wedge r \le b. & \\
\qquad\quad \mathit{disjE} \cdot r = b \cdot r \ne b \cdot \exists\, x\ q.\ \mathit{Suc}\ n = \mathit{Suc}\ b * q + x \wedge x \le b \bullet & \\
\qquad\quad (\mathit{nat\text{-}eq\text{-}dec} \cdot r \cdot b) \bullet & \\
\qquad\quad (\boldsymbol{\lambda} \mathit{Ha}\colon r = b. & \\
\qquad\qquad \mathit{exI} \cdot \mathit{TYPE}(\mathit{nat}) \cdot (\lambda x.\ \exists\, q.\ \mathit{Suc}\ n = \mathit{Suc}\ b * q + x \wedge x \le b) \cdot \mathit{0} \bullet & \longleftarrow \text{remainder} \\
\qquad\qquad (\mathit{exI} \cdot \mathit{TYPE}(\mathit{nat}) \cdot (\lambda x.\ \mathit{Suc}\ n = \mathit{Suc}\ b * x + \mathit{0} \wedge \mathit{0} \le b) \cdot \mathit{Suc}\ q \bullet & \longleftarrow \text{quotient} \\
\qquad\qquad\quad (\mathit{conjI} \cdot \mathit{Suc}\ n = \mathit{Suc}\ b * \mathit{Suc}\ q + \mathit{0} \cdot \mathit{0} \le b \bullet & \\
\qquad\qquad\qquad (\mathit{subst} \cdot \mathit{TYPE}(\mathit{nat}) \cdot \mathit{Suc}\ (q + b * q + b) \cdot \mathit{Suc}\ n \cdot & \\
\qquad\qquad\qquad\quad (\lambda z.\ z = \mathit{Suc}\ b * \mathit{Suc}\ q + \mathit{0}) \bullet & \\
\qquad\qquad\qquad\quad \cdots \bullet & \\
\qquad\qquad\qquad\quad \cdots) \bullet & \\
\qquad\qquad\qquad (\mathit{le0} \cdot b))\ )) \bullet & \\
\qquad\quad (\boldsymbol{\lambda} \mathit{Ha}\colon r \ne b. & \\
\qquad\qquad \mathit{exI} \cdot \mathit{TYPE}(\mathit{nat}) \cdot (\lambda x.\ \exists\, q.\ \mathit{Suc}\ n = \mathit{Suc}\ b * q + x \wedge x \le b) \cdot \mathit{Suc}\ r \bullet & \longleftarrow \text{remainder} \\
\qquad\qquad (\mathit{exI} \cdot \mathit{TYPE}(\mathit{nat}) \cdot (\lambda x.\ \mathit{Suc}\ n = \mathit{Suc}\ b * x + \mathit{Suc}\ r \wedge \mathit{Suc}\ r \le b) \cdot q \bullet & \longleftarrow \text{quotient} \\
\qquad\qquad\quad (\mathit{conjI} \cdot \mathit{Suc}\ n = \mathit{Suc}\ b * q + \mathit{Suc}\ r \cdot \mathit{Suc}\ r \le b \bullet & \\
\qquad\qquad\qquad (\mathit{subst} \cdot \mathit{TYPE}(\mathit{nat}) \cdot \mathit{Suc}\ (q + b * q + r) \cdot \mathit{Suc}\ n \cdot & \\
\qquad\qquad\qquad\quad (\lambda z.\ z = \mathit{Suc}\ b * q + \mathit{Suc}\ r) \bullet & \\
\qquad\qquad\qquad\quad \cdots \bullet & \\
\qquad\qquad\qquad\quad \cdots) \bullet & \\
\qquad\qquad\qquad (\mathit{ccontr} \cdot \mathit{Suc}\ r \le b \bullet & \\
\qquad\qquad\qquad\quad (\boldsymbol{\lambda} \mathit{Hb}\colon \neg\ \mathit{Suc}\ r \le b. & \\
\qquad\qquad\qquad\qquad \mathit{notE} \cdot \mathit{Suc}\ \mathit{0} = \mathit{0} \cdot \mathit{False} \bullet (\mathit{Suc\text{-}not\text{-}Zero} \cdot \mathit{0}) \bullet & \\
\qquad\qquad\qquad\qquad\ \cdots \bullet & \\
\qquad\qquad\qquad\qquad\ \cdots)))\ )) &
\end{array}
$$

Figure 5.2: Primitive proof of existence of quotient and remainder

```
equal-elim · · · · · · · · •
 (symmetric · · · · · · · · •
   (combination · TYPE(prop) · TYPE(nat ⇒ nat ⇒ nat × nat) ·
     (λx. a = Suc b ∗ snd (x a b) + fst (x a b) ∧ fst (x a b) ≤ b) · · · · ·
     division ·
     (λx xa. nat-rec (0, 0)
              (λn H. case H of (x, y) ⇒
                        case nat-eq-dec x xa of Left ⇒ (0, Suc y) | Right ⇒ (Suc x, y)) x) •
     (reflexive · · · · • division-def)) •                    ⟵ expand the definition of division
 (induct-P-correctness · TYPE(nat × nat) ·
   (λx xa. xa = Suc b ∗ snd x + fst x ∧ fst x ≤ b) · (0, 0) ·
   (λn H. case H of
          (x, y) ⇒ case nat-eq-dec x b of Left ⇒ (0, Suc y) | Right ⇒ (Suc x, y)) · a •
   (exI-correctness · TYPE(nat) · TYPE(nat) · (λx xa. 0 = Suc b ∗ x + xa ∧ xa ≤ b) · 0 · 0 •
     (conjI · 0 = Suc b ∗ 0 + 0 · 0 ≤ b •
       (subst · TYPE(nat) · 0 · Suc b ∗ 0 + 0 · (λx. 0 = x) •
         · · · •
         (HOL.refl · TYPE(nat) · 0)) •
       (le0 · b)) ) •
   (λn H Ha: n = Suc b ∗ snd H + fst H ∧ fst H ≤ b.
     exE-correctness · TYPE(nat × nat) · TYPE(nat) · TYPE(nat) ·
      (λx xa. n = Suc b ∗ x + xa ∧ xa ≤ b) · H ·
      (λx. Suc n = Suc b ∗ snd x + fst x ∧ fst x ≤ b) ·
      (λr Ha. case nat-eq-dec r b of Left ⇒ (0, Suc Ha) | Right ⇒ (Suc r, Ha)) • Ha •
      (λr Ha H: n = Suc b ∗ Ha + r ∧ r ≤ b.
        disjE-correctness3 · TYPE(nat × nat) · r = b · r ≠ b · nat-eq-dec r b ·
         (λx. Suc n = Suc b ∗ snd x + fst x ∧ fst x ≤ b) · (0, Suc Ha) · (Suc r, Ha) •
         (nat-eq-dec-correctness · r · b) •
         (λHb: r = b.
           exI-correctness · TYPE(nat) · TYPE(nat) ·
            (λx xa. Suc n = Suc b ∗ x + xa ∧ xa ≤ b) · Suc Ha · 0 •
            (conjI · Suc n = Suc b ∗ Suc Ha + 0 · 0 ≤ b •
              (subst · TYPE(nat) · Suc (Ha + b ∗ Ha + b) · Suc n ·
                (λx. x = Suc b ∗ Suc Ha + 0) •
                · · · • · · ·) •
              (le0 · b)) ) •
         (λHb: r ≠ b.
           exI-correctness · TYPE(nat) · TYPE(nat) ·
            (λx xa. Suc n = Suc b ∗ x + xa ∧ xa ≤ b) · Ha · Suc r •
            (conjI · Suc n = Suc b ∗ Ha + Suc r · Suc r ≤ b •
              (subst · TYPE(nat) · Suc (Ha + b ∗ Ha + r) · Suc n ·
                (λx. x = Suc b ∗ Ha + Suc r) •
                · · · • · · ·) •
              (ccontr · Suc r ≤ b •
                (λHc: ¬ Suc r ≤ b.
                  notE · Suc 0 = 0 · False • (Suc-not-Zero · 0) •
                   · · · • · · ·))) ))))
```

Figure 5.3: Correctness proof for quotient and remainder

```
datatype nat = id0 | Suc of nat;

fun nat_rec f1 f2 id0 = f1
  | nat_rec f1 f2 (Suc nat) = f2 nat (nat_rec f1 f2 nat);

datatype sumbool = Left | Right;

fun nat_eq_dec x =
  (fn xa =>
    nat_rec (fn x => (case x of id0 => Left | Suc x => Right))
      (fn x => fn H2 => fn xa =>
        (case xa of id0 => Right
          | Suc x => (case H2 x of Left => Left | Right => Right)))
      x xa);

fun division x =
  (fn xa =>
    nat_rec (id0, id0)
      (fn n => fn H =>
        (case H of
          (x, xb) =>
            (case nat_eq_dec x xa of Left => (id0, Suc xb)
              | Right => (Suc x, xb))))
      x);
```

Figure 5.4: ML code generated by Isabelle for division function

$$\begin{array}{l}
\textit{nat-eq-dec} \equiv \\
\lambda x\ xa.\ \textit{nat-rec}\ (\lambda x.\ \textit{case}\ x\ \textit{of}\ 0 \Rightarrow \textit{Left} \mid \textit{Suc}\ x \Rightarrow \textit{Right}) \\
\qquad\quad (\lambda x\ H2\ xa. \\
\qquad\qquad \textit{case}\ xa\ \textit{of}\ 0 \Rightarrow \textit{Right} \mid \textit{Suc}\ \textit{nat} \Rightarrow \textit{case}\ H2\ \textit{nat}\ \textit{of}\ \textit{Left} \Rightarrow \textit{Left} \mid \textit{Right} \Rightarrow \textit{Right}) \\
\qquad\quad x\ xa
\end{array}$$

In order to keep the extracted program modular, the function *division* contains just a reference to the auxiliary function *nat-eq-dec*. From the Isabelle/HOL definition of the *division* function shown above, Isabelle's code generator can automatically generate an executable ML program which, together with the required auxiliary functions, is shown in Figure 5.4.
The correctness theorem corresponding to the *division* function is

$$a = \textit{Suc}\ b * \textit{snd}\ (\textit{division}\ a\ b) + \textit{fst}\ (\textit{division}\ a\ b) \wedge \textit{fst}\ (\textit{division}\ a\ b) \le b$$

The proof of this theorem, which is shown in Figure 5.3, is automatically derived by the program extraction module by transforming the original proof using the function corr introduced in §4.2.3. The correctness proof uses the theorems

$$\begin{array}{ll}
\textit{exI-correctness:} & P^R\ y\ x \Longrightarrow P^R\ (\textit{snd}\ (x,\ y))\ (\textit{fst}\ (x,\ y)) \\
\textit{exE-correctness:} & P^R\ (\textit{snd}\ p)\ (\textit{fst}\ p) \Longrightarrow \\
 & (\bigwedge x\ y.\ P^R\ y\ x \Longrightarrow Q^R\ (f\ x\ y)) \Longrightarrow Q^R\ (\textit{case}\ p\ \textit{of}\ (x,\ y) \Rightarrow f\ x\ y) \\
\textit{disjE-correctness3:} & \textit{case}\ x\ \textit{of}\ \textit{Left} \Rightarrow P^R \mid \textit{Right} \Rightarrow Q^R \Longrightarrow \\
 & (P^R \Longrightarrow R^R\ f) \Longrightarrow (Q^R \Longrightarrow R^R\ g) \Longrightarrow R^R\ (\textit{case}\ x\ \textit{of}\ \textit{Left} \Rightarrow f \mid \textit{Right} \Rightarrow g)
\end{array}$$

introduced in §4.3.3, which assert the correctness of the programs corresponding to the basic inference rules *exI*, *exE*, and *disjE*, as well as the correctness theorem

$$\begin{array}{ll} \textit{induct-P-correctness:} & P^R\ f\ 0 \Longrightarrow \\ & (\bigwedge nat\ rnat.\ P^R\ rnat\ nat \Longrightarrow P^R\ (g\ nat\ rnat)\ (Suc\ nat)) \Longrightarrow \\ & P^R\ (nat\text{-}rec\ f\ g\ n)\ n \end{array}$$

for the program corresponding to the induction principle on natural numbers, which has been introduced in §4.3.4. Moreover, since the correctness proof is modular, it also relies on the correctness of the program extracted from the proof of decidability of equality on natural numbers, which is expressed by the theorem

$$\textit{nat-eq-dec-correctness:}\quad \textit{case nat-eq-dec } m\ n \textit{ of Left} \Rightarrow m = n \mid \textit{Right} \Rightarrow m \neq n$$

5.1.2 Comparison with Coq

To examine how Isabelle compares with a system based on dependent type theory, we now reformulate the proof from the previous section in the Coq system [12]. As far as program extraction is concerned, the main difference between Isabelle and Coq is the treatment of computationally interesting and noninteresting objects. Isabelle's program extraction mechanism distinguishes between *relevant* and *irrelevant* predicate variables as described in §4.2. It is important to note that this information about the computational relevance of objects is not part of Isabelle's logic itself, but is an extra-logical concept. This is in contrast to the Coq system based on the Calculus of Inductive Constructions, where this information is encoded into the type system. The type system of Coq has two universes `Set` and `Prop`, which are inhabited by computationally interesting and computationally noninteresting types, respectively. Note that a type which is an inhabitant of `Set` cannot be an inhabitant of `Prop`, and vice versa. As a consequence, the Coq library contains several versions of each logical operator, each having a different computational content. As has already been mentioned in §4.3.5.3, most logical operators in Coq are defined inductively. For example, the definitions of the most frequently used variants of the existential quantifier look as follows:

```
Inductive ex [A:Set;P:A->Prop] : Prop
    := ex_intro : (x:A)(P x)->(ex A P).

Inductive sig [A:Set;P:A->Prop] : Set
    := exist : (x:A)(P x) -> (sig A P).

Inductive sigS [A:Set;P:A->Set] : Set
    := existS : (x:A)(P x) -> (sigS A P).
```

The syntax for `ex`, `sig` and `sigS` is `(EX x | (P x))`, `{x:A | (P x)}` and `{x:A & (P x)}`, respectively. The `ex` quantifier resides in the `Prop` universe and therefore has no computational content at all. The `sig` quantifier resides in the `Set` universe, but its body `(P x)` is an inhabitant of the `Prop` universe. This variant of the existential quantifier can therefore be used if one is just interested in the existential witness, but not in the computational content of the body of the quantifier. Finally, the `sigS` quantifier resides in the `Set` universe, too, but in contrast to the `sig` quantifier, its body `(P x)` is an inhabitant of `Set` as well, i.e. this variant of the existential quantifier is useful if one is interested in both the witness and the computational content of the body of the quantifier. Note that the type `A` of the witness `x` always resides in the `Set` universe, since this will usually be some kind of datatype, such as natural numbers or lists, i.e. a computationally relevant type. Things are similar for disjunction, for which Coq offers the three variants

```
Inductive or [A,B:Prop] : Prop :=
     or_introl : A -> (or A B)
   | or_intror : B -> (or A B).

Inductive sumbool [A,B:Prop] : Set
    := left  : A -> (sumbool A B)
     | right : B -> (sumbool A B).

Inductive sumor [A:Set;B:Prop] : Set
    := inleft  : A -> (sumor A B)
     | inright : B -> (sumor A B).
```

with syntax `A \/ B`, `{A}+{B}` and `A+{B}`, respecively.

The approach taken in Coq of marking the computational content via the type system was found to be unsuitable for Isabelle, since the introduction of another type of truth values, say bool′, for formulae with computational content, in addition to the already existing type bool, would have made it impossible to use already existing theorems from standard libraries, which are part of the Isabelle distribution. Apart from the necessity to redefine logical operators and to reformulate many existing theorems for the type bool′, the introduction of a new type of truth values would have been likely to confuse users not familiar with program extraction and to interfere with existing applications. In contrast, the program extraction framework introduced in §4.2 does not require a change of the very foundations of Isabelle's meta and object logics. Instead of the two quantifiers `sig` and `sigS` above, just one quantifier is needed in Isabelle, since one can deduce from the structure of the formula in the body of the quantifier, i.e. via the *type extraction* mechanism described in §4.2.1, which of the two variants is meant. A counterpart of Coq's `ex` quantifier withouth any computational content, though, is not available in Isabelle/HOL by default.

We now get back to the task of proving the existence of a remainder `r` and quotient `q` for a dividend `a` and divisor `b`. To write down the specification, we need two of the existential quantifiers introduced above. The outer quantifier has the form `{r:nat & ...}`, since its body contains another quantifier with computational content. The inner quantifier, however, has the form `{q:nat | ...}`, since its body contains just a specification of the properties of `r` and `q`, which does not contribute to the computational content. The whole theorem is stated in Coq as follows:

```
(a,b:nat){r:nat & {q:nat | a=(plus (mult (S b) q) r) /\ (le r b)}}
```

Here, `(x:P)(...)` is a *dependent product*, which plays the role of a universal quantifier. The script for proving this theorem in Coq[1] is shown in Figure 5.5. The idea underlying the proof is almost the same as for the Isar proof description presented in §5.1, although it is much less readable than its Isar counterpart. Coq also does not offer as much automation as Isabelle, which is why some of the rewrite rules needed in the proof have to be applied manually. The primitive proof term, which is built by Coq behind the scenes when executing the above proof script, is shown in Figure 5.6. Terms of the form `[x:P](...)` occurring in the proof denote λ-abstractions. Again, as in the Isabelle version of the proof object shown in Figure 5.2, the shaded subproofs are computationally irrelevant. In Coq, a subproof p is computationally irrelevant if $\Gamma \vdash p : P$ and $\Gamma \vdash P :$ `Prop`. Functions with names of the form "`..._rec`" or "`..._ind`" appearing in the proof correspond to *elimination rules*. The "`..._rec`" versions, such

[1] This script has been tested with Coq 7.3.1 (October 2002)

```
Theorem division : (a,b:nat){r:nat & {q:nat | a=(plus (mult (S b) q) r) /\ (le r b)}}.
Intros.
Elim a.
Exists O; Exists O.
Rewrite <- (mult_n_O (S b)).
Auto with arith.
Intros n H1.
Elim H1; Intros r H2.
Elim H2; Intros q H3.
Elim H3.
Elim (eq_nat_dec r b).
Exists O; Exists (S q).
Rewrite <- mult_n_Sm.
Rewrite <- plus_n_O.
Rewrite <- plus_n_Sm.
Rewrite -> a0 in H.
Auto with arith.
Exists (S r); Exists q.
Rewrite <- plus_n_Sm.
Elim (le_lt_or_eq r b H0).
Auto.
Tauto.
Qed.
```

Figure 5.5: Coq proof script for existence of quotient and remainder

as `sig_rec`, `sigS_rec` and `sumbool_rec`, which correspond to an elimination of the quantifiers `{x:A | (P x)}` and `{x:A & (P x)}`, and of the computationally relevant disjunction `{A}+{B}`, respectively, prove computationally *relevant* statements, whereas the "`..._ind`" versions, such as `eq_ind`, which essentially corresponds to the substitution rule, prove computationally *irrelevant statements*. Coq's type system enforces that a statement *with* computational content may not be proved by applying an elimination rule to a proof of a statement *without* computational content. For example, we may not prove a computationally relevant existential statement of the form `{x:A | (P x)}` by applying an elimination rule to a proof of a computationally irrelevant existential statement of the form `(EX x | (P x))`, i.e. there is only an `ex_ind`, but no `ex_rec` rule. Intuitively, this makes sure that the parts of a proof which correspond to computations do not rely on subproofs for which no program can be extracted, and subproofs of statements which are in `Prop` can safely be deleted during extraction. In particular, it is even safe to assume a classical axiom of the form

```
  Axiom classic: (P:Prop)(P \/ ~(P)).
```

However, the variant

```
  Axiom unsafe_classic: (P:Prop)({P}+{~P})
```

of the above axiom involving the computationally relevant disjunction is *unsafe* and must not be assumed, since there is no program corresponding to this axiom. Worse yet, as has been shown by Geuvers [40] recently, assuming this axiom even makes the logic inconsistent, i.e. one can produce a proof of `False` using this axiom.

```
[a,b:nat]
 (nat_rec
   [n:nat]{r:nat & {q:nat | (n=(plus (mult (S b) q) r)/\(le r b))}}
   (existS nat [r:nat]{q:nat | ((0)=(plus (mult (S b) q) r)/\(le r b))} (0)
     (exist nat [q:nat](0)=(plus (mult (S b) q) (0))/\(le (0) b) (0)
       (eq_ind nat (0) [n:nat](0)=(plus n (0))/\(le (0) b)
         <((0)=(plus (0) (0))),(le (0) b)>{(plus_n_O (0)),(le_O_n b)}
         (mult (S b) (0)) (mult_n_O (S b))) ))
   [n:nat;
    H1:({r:nat & {q:nat | (n=(plus (mult (S b) q) r)/\(le r b))}})]
    (sigS_rec nat
      [r:nat]{q:nat | (n=(plus (mult (S b) q) r)/\(le r b))}
      [_:({r:nat & {q:nat | (n=(plus (mult (S b) q) r)/\(le r b))}})]
       {r:nat & {q:nat | ((S n)=(plus (mult (S b) q) r)/\(le r b))}}
      [r:nat; H2:({q:nat | (n=(plus (mult (S b) q) r)/\(le r b))})]
       (sig_rec nat [q:nat]n=(plus (mult (S b) q) r)/\(le r b)
         [_:({q:nat | (n=(plus (mult (S b) q) r)/\(le r b))})]
          {r0:nat & {q:nat | ((S n)=(plus (mult (S b) q) r0)/\(le r0 b))}}
         [q:nat; H3:(n=(plus (mult (S b) q) r)/\(le r b))]
          (and_rec n=(plus (mult (S b) q) r) (le r b)
            {r0:nat & {q0:nat | ((S n)=(plus (mult (S b) q0) r0)/\(le r0 b))}}
            (sumbool_rec r=b ~r=b
              [_:({r=b}+{~r=b})]
               n=(plus (mult (S b) q) r)
               ->(le r b)
               ->{r0:nat & {q0:nat | ((S n)=(plus (mult (S b) q0) r0)/\(le r0 b))}}
              [a0:(r=b); H:(n=(plus (mult (S b) q) r)); _:(le r b)]
               (existS nat
                 [r0:nat]
                  {q0:nat | ((S n)=(plus (mult (S b) q0) r0)/\(le r0 b))} (0)
                 (exist nat
                   [q0:nat](S n)=(plus (mult (S b) q0) (0))/\(le (0) b) (S q)
                   (eq_ind nat (plus (mult (S b) q) (S b))
                     [n0:nat](S n)=(plus n0 (0))/\(le (0) b)
                     ...
                     (mult (S b) (S q)) (mult_n_Sm (S b) q)) ))
              [b0:(~r=b); H:(n=(plus (mult (S b) q) r)); H0:(le r b)]
               (existS nat
                 [r0:nat]
                  {q0:nat | ((S n)=(plus (mult (S b) q0) r0)/\(le r0 b))} (S r)
                 (exist nat
                   [q0:nat](S n)=(plus (mult (S b) q0) (S r))/\(le (S r) b) q
                   (eq_ind nat (S (plus (mult (S b) q) r))
                     [n0:nat](S n)=n0/\(le (S r) b)
                     ...
                     (plus (mult (S b) q) (S r))
                     (plus_n_Sm (mult (S b) q) r)) )) (eq_nat_dec r b))
            H3) H2) H1) a)
```

Figure 5.6: Coq proof term for existence of quotient and remainder

```
type nat =
  | O
  | S of nat

type ('a, 'p) sigS =
  | ExistS of 'a * 'p

type sumbool =
  | Left
  | Right

let rec eq_nat_dec n m =
  match n with
    | O -> (match m with
              | O -> Left
              | S n0 -> Right)
    | S n0 -> (match m with
                 | O -> Right
                 | S n1 -> eq_nat_dec n0 n1)

let rec division a b =
  match a with
    | O -> ExistS (O, O)
    | S n ->
        let ExistS (x, x0) = division n b in
        (match eq_nat_dec x b with
           | Left -> ExistS (O, (S x0))
           | Right -> ExistS ((S x), x0))
```

Figure 5.7: OCaml code generated by Coq from proof of `division` theorem

Since a distinction of computationally relevant and irrelevant objects via the type system is not possible in Isabelle, one may actually use the Isabelle/HOL rule

$$\textit{excluded-middle} : \neg P \vee P$$

or the *ccontr* rule used in the proof shown in Figure 5.2, to prove a computationally relevant statement. However, the evil hour comes when running the extraction function, which will terminate with an error when applied to such a proof, whereas in Coq we would not have been able to *construct* such a proof in the first place.

In contrast to Isabelle, which first defines the extracted program as a function inside the logic, Coq directly generates OCaml code from proofs. Figure 5.7 shows the code which is generated by Coq from the proof shown in Figure 5.6. Note that the Coq counterpart of the recursion combinator `nat_rec`, which appears in the ML code shown in Figure 5.4, has been unfolded in the above program, which makes it a bit more readable.

5.2 Warshall's algorithm

As a larger example, we show how Warshall's algorithm for computing the transitive closure of a relation can be derived using program extraction. The formalization is inspired by Berger et al. [19]. It has also been treated in the Coq system [12] by Paulin-Mohring [86]. In the sequel, a relation will be a function mapping two elements of a type to a boolean value.

datatype $b = T \mid F$
types $'a\ rel = {'a} \Rightarrow {'a} \Rightarrow b$

To emphasize that the relation has to be *decidable*, we use the datatype b instead of the built-in type *bool* of HOL for this purpose.

In order to write down the specification of the algorithm, it will be useful to introduce a function *is-path'*, where *is-path'* $r\ x\ ys\ z$ holds iff there is a path from x to z with intermediate nodes ys with respect to a relation r.

consts $\mathit{is\text{-}path'} :: {'a}\ rel \Rightarrow {'a} \Rightarrow {'a}\ list \Rightarrow {'a} \Rightarrow bool$
primrec
$\mathit{is\text{-}path'}\ r\ x\ []\ z = (r\ x\ z = T)$
$\mathit{is\text{-}path'}\ r\ x\ (y \# ys)\ z = (r\ x\ y = T \wedge \mathit{is\text{-}path'}\ r\ y\ ys\ z)$

Paths will be modeled as triples consisting of a source node, a list of intermediate nodes and a target node. In the sequel, nodes will be natural numbers. Using the auxiliary function *is-path'* we can now define a function *is-path*, where *is-path* $r\ p\ i\ j\ k$ holds iff p is a path from j to k with intermediate nodes less than i. For brevity, a path with this property will be called an i-path. We also introduce a function *conc* for concatenating two paths.

constdefs
$\mathit{is\text{-}path} :: nat\ rel \Rightarrow (nat \times nat\ list \times nat) \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow bool$
$\mathit{is\text{-}path}\ r\ p\ i\ j\ k \equiv fst\ p = j \wedge snd\ (snd\ p) = k \wedge$
$\quad \mathit{list\text{-}all}\ (\lambda x.\ x < i)\ (fst\ (snd\ p)) \wedge \mathit{is\text{-}path'}\ r\ (fst\ p)\ (fst\ (snd\ p))\ (snd\ (snd\ p))$
$conc :: ({'a} \times {'a}\ list \times {'a}) \Rightarrow ({'a} \times {'a}\ list \times {'a}) \Rightarrow ({'a} \times {'a}\ list \times {'a})$
$conc\ p\ q \equiv (fst\ p,\ fst\ (snd\ p)\ @\ fst\ q \# fst\ (snd\ q),\ snd\ (snd\ q))$

The main proof relies on several lemmas about properties of *is-path*. For example, if p is an i-path from j to k, then p is also a *Suc* i-path.

lemma *lemma1*: $\bigwedge p.\ \mathit{is\text{-}path}\ r\ p\ i\ j\ k \Longrightarrow \mathit{is\text{-}path}\ r\ p\ (Suc\ i)\ j\ k$

If p is a *0*-path from j to k, then relation r has an edge connecting j and k.

lemma *lemma2*: $\bigwedge p.\ \mathit{is\text{-}path}\ r\ p\ 0\ j\ k \Longrightarrow r\ j\ k = T$

If p is an i-path from j to i, and q is an i-path from i to k, then concatenating these paths yields a *Suc* i-path from j to k.

lemma *lemma3*: $\bigwedge p\ q.\ \mathit{is\text{-}path}\ r\ p\ i\ j\ i \Longrightarrow \mathit{is\text{-}path}\ r\ q\ i\ i\ k \Longrightarrow$
$\quad \mathit{is\text{-}path}\ r\ (conc\ p\ q)\ (Suc\ i)\ j\ k$

The last lemma is central to the proof of the main theorem. It says that if there is a *Suc* i-path from j to k, but no i-path, then there must be i-paths from j to i and from i to k.

lemma *lemma4*: $\bigwedge p.\ \mathit{is\text{-}path}\ r\ p\ (Suc\ i)\ j\ k \Longrightarrow \neg\ \mathit{is\text{-}path}\ r\ p\ i\ j\ k \Longrightarrow$
$\quad (\exists\, q.\ \mathit{is\text{-}path}\ r\ q\ i\ j\ i) \wedge (\exists\, q.\ \mathit{is\text{-}path}\ r\ q\ i\ i\ k)$

The first component of the conjunction can be proved by induction on the list of intermediate nodes of path p. The proof of the second component is symmetric to the proof of the first component, using "reverse induction". Although this lemma can be proved constructively, its computational content is not used in the main theorem. To emphasize this, we rephrase it, writing $\neg\ (\forall\, x.\ \neg\ P\ x)$ instead of $\exists\, x.\ P\ x$.

```
theorem warshall: ⋀j k. ¬ (∃p. is-path r p i j k) ∨ (∃p. is-path r p i j k)
proof (induct i)
  case (0 j k) show ?case — induction basis
  proof (cases r j k)
    assume r j k = T
    hence is-path r (j, [], k) 0 j k by (simp add: is-path-def)
    hence ∃p. is-path r p 0 j k .. thus ?thesis ..
  next
    assume r j k = F hence r j k ≠ T by simp
    hence ¬ (∃p. is-path r p 0 j k) by (rules dest: lemma2) thus ?thesis ..
  qed
next
  case (Suc i j k) thus ?case — induction step
  proof
    assume ∃p. is-path r p i j k
    hence ∃p. is-path r p (Suc i) j k by (rules intro: lemma1) thus ?case ..
  next
    assume h1: ¬ (∃p. is-path r p i j k)
    from Suc show ?case
    proof
      assume ¬ (∃p. is-path r p i j i)
      with h1 have ¬ (∃p. is-path r p (Suc i) j k) by (rules dest: lemma4')
      thus ?case ..
    next
      assume ∃p. is-path r p i j i
      then obtain p where h2: is-path r p i j i ..
      from Suc show ?case
      proof
        assume ¬ (∃p. is-path r p i i k)
        with h1 have ¬ (∃p. is-path r p (Suc i) j k) by (rules dest: lemma4')
        thus ?case ..
      next
        assume ∃q. is-path r q i i k
        then obtain q where is-path r q i i k ..
        with h2 have is-path r (conc p q) (Suc i) j k by (rule lemma3)
        hence ∃pq. is-path r pq (Suc i) j k .. thus ?case ..
      qed
    qed
  qed
qed
```

Figure 5.8: Warshall's algorithm formalized in Isabelle/Isar

```
lemma lemma4': ⋀p. is-path r p (Suc i) j k ⟹ ¬ is-path r p i j k ⟹
  ¬ (∀q. ¬ is-path r q i j i) ∧ ¬ (∀q. ¬ is-path r q i i k)
```

The main theorem can now be stated as follows: For a given relation r, for all i and for every two nodes j and k there either exists an i-path p from j to k, or no such path exists. Of course, this would be trivial to prove classically. However, a constructive proof of this statement actually yields a function that either returns *Some p* if there is a path or returns *None* otherwise.

The proof is by induction on i. In the base case, we have to find a 0-path from j to k, which can only exist if r has an edge connecting these two nodes. Otherwise there can be no such path according to *lemma2*. In the step case, we are supposed to find a *Suc* i-path from j to k. By appeal to the induction hypothesis, we can decide if we already have an i-path from j to k. If this is the case, we can easily conclude by *lemma1* that this is also a *Suc* i-path. Otherwise, by appealing to the induction hypothesis two more times, we check whether we have i-paths from j to i and from i to k. If there are such paths, we combine them to get a *Suc* i-path from j to k by *lemma3*. Otherwise, if there is no i-path from j to i or from i to k, there can be no *Suc* i-path from j to k either, because this would contradict *lemma4'*. The formalization of this proof in Isabelle/Isar is shown in Figure 5.8. From this proof, the following program is extracted by Isabelle:

```
warshall ≡
λr i j k.
  nat-rec (λi j. case r i j of T ⇒ Some (i, [], j) | F ⇒ None)
   (λk H i j.
      case H i j of
      None ⇒
        case H i k of None ⇒ None
        | Some p ⇒ case H k j of None ⇒ None | Some q ⇒ Some (conc p q)
      | Some q ⇒ Some q)
   i j k
```

Applying the definition of realizability presented in §4.3 yields the following correctness theorem, which is automatically derived from the above proof:

case warshall r i j k of None $\Rightarrow \forall x.\ \neg$ *is-path r x i j k* | *Some q* $\Rightarrow$ *is-path r q i j k*

5.3 Higman's lemma

Higman's lemma [50] is an interesting problem from the field of combinatorics. It can be considered as a specific instance of Kruskal's famous tree theorem, which is useful for proving the termination of term rewriting systems using so-called *simplification orders*. Higman's lemma states that every infinite sequence of words $(w_i)_{0\leq i<\omega}$ contains two words w_i and w_j with $i < j$ such that w_i can be *embedded* into w_j. A sequence with this property is also called *good*, otherwise *bad*. Although a quite elegant *classical* proof of this statement has been given by Nash-Williams [73] using a so-called *minimal bad sequence argument*, there has been a growing interest in obtaining *constructive* proofs of Higman's lemma recently. This is due to the additional informative content inherent in constructive proofs. For example, a termination proof of a string rewrite system based on a constructive proof of Higman's lemma could be used to obtain upper bounds on the length of reduction sequences.

The first formalization of Higman's lemma using a theorem prover was done by Murthy [71] in the Nuprl system [27]. Murthy first formalized Nash-Williams' classical proof, then translated it into a constructive proof using a *double negation translation* followed by Friedman's *A-translation* and finally extracted a program from the resulting proof. Unfortunately, although correct in principle, the program obtained in this way was so huge that it was both incomprehensible and impossible to execute within a reasonable amount of time even on the fastest computing equipment available. This rather disappointing experience prompted several scientists to think about direct formalizations of constructive proofs of Higman's lemma, notably Murthy and Russell [72], as well as Fridlender [38], who formalized Higman's lemma using the ALF proof editor [65] based on Martin-Löf's type theory. Fridlender's paper also gives a detailed account of the history of Higman's lemma. An excellent overview of various different formalizations of Higman's lemma is given by Seisenberger [108]. A particularly elegant and short constructive proof, based entirely on inductive definitions, has been suggested by Coquand and Fridlender [29]. This proof, which has also been formalized by Seisenberger [108] in the Minlog theorem prover, will now be used as a test case for Isabelle's program extraction module. In contrast to the rather abstract exposition given by Coquand and Fridlender, we also try to give an intuitive graphical description of the computational behaviour of the extracted programs.

We start with a few basic definitions. Words are modelled as lists of letters from the two letter alphabet[2]

datatype *letter* $= A \mid B$

The embedding relation on words is defined inductively as follows:

consts *emb* :: (*letter list* × *letter list*) *set*
inductive *emb*
intros
  emb0: $[] \trianglelefteq bs$
  emb1: $as \trianglelefteq bs \Longrightarrow as \trianglelefteq b \# bs$
  emb2: $as \trianglelefteq bs \Longrightarrow a \# as \trianglelefteq a \# bs$

Intuitively, a word *as* can be embedded into a word *bs*, if we can obtain *as* by deleting letters from *bs*. For example, $[A, A] \trianglelefteq [B, A, B, A]$. In order to formalize the notion of a good sequence, it is useful to define the set $L\ v$ of all lists of words containing a word which can be embedded into v:

consts *L* :: *letter list* ⇒ *letter list list set*
inductive *L v*
intros
  L0: $w \trianglelefteq v \Longrightarrow w \# ws \in L\ v$
  L1: $ws \in L\ v \Longrightarrow w \# ws \in L\ v$

A list of words is *good* if its tail is either *good* or contains a word which can be embedded into the word occurring at the head position of the list:

consts *good* :: *letter list list set*
inductive *good*
intros
  good0: $ws \in L\ w \Longrightarrow w \# ws \in good$
  good1: $ws \in good \Longrightarrow w \# ws \in good$

In contrast to Coquand [29], who defines *Cons* such that it appends elements to the right of the list, we use the usual definition of *Cons*, which appends elements to the left. Therefore, the predicates on lists of words defined in this section, such as the *good* predicate introduced above work "in the opposite direction", e.g. $[[A, A], [A, B], [B]] \in good$, since $[B] \trianglelefteq [A, B]$. In order to express the fact that every infinite sequence is good, we define a predicate *bar* as follows:

consts *bar* :: *letter list list set*
inductive *bar*

[2]It is worth noting that the extension of the proof to an arbitrary finite alphabet is not at all trivial. For details, see Seisenberger's PhD. thesis [109].

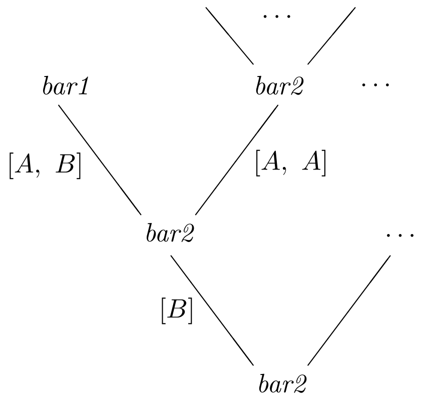


Figure 5.9: Computational content of *bar*

intros

$bar1$: $ws \in good \Longrightarrow ws \in bar$
$bar2$: $(\bigwedge w.\ w \ \# \ ws \in bar) \Longrightarrow ws \in bar$

Intuitively, $ws \in bar$ means that either the list of words ws is already *good*, or successively adding words will turn it into a good list. Consequently, $[] \in bar$ means that every infinite sequence $(w_i)_{0 \le i < \omega}$ must be good, i.e. have a prefix $w_0 \ \dots \ w_n$ with $[w_n,\ \dots,\ w_0] \in good$, since by successively adding words $w_0,\ w_1,\ \dots$ to the empty list, we must eventually arrive at a list which is good. Note that the above definition of *bar* is closely related to Brouwer's more general principle of *bar induction* [116, Chapter 4, §8]. Like the accessible part of a relation defined in §4.3.5.3, the definition of *bar* embodies a kind of well-foundedness principle. The datatype

$$\textsf{datatype}\ barT = bar1\ (letter\ list\ list)\ |\ bar2\ (letter\ list\ list)\ (letter\ list \Rightarrow barT)$$

representing the computational content of $ws \in bar$ is an infinitely branching tree from which one can read off how many words have to be appended to the sequence of words ws in order to turn it into a *good* sequence. The branches of this tree are labelled with words. For each appended word w, one moves one step closer to the leaves of the tree, following the branch labelled with w. When a leaf, i.e. the constructor *bar1* is reached, the resulting sequence of words must be *good*. An example for such a tree is shown in Figure 5.9. The realizability predicate for *bar* is characterized by the introduction rules

$ws \in good \Longrightarrow (bar1\ ws,\ ws) \in barR$
$(\bigwedge w.\ (f\ w,\ w \ \# \ ws) \in barR) \Longrightarrow (bar2\ ws\ f,\ ws) \in barR$

This means that if $(bar2\ ws\ f_0,\ ws) \in barR$ and

$$f_0\ w_0 = bar2\ (w_0 \ \# \ ws)\ f_1, \quad f_1\ w_1 = bar2\ (w_1 \ \# \ w_0 \ \# \ ws)\ f_2, \quad \dots,$$
$$f_{n-1}\ w_{n-1} = bar2\ (w_{n-1} \ \# \ \cdots \ \# \ w_0 \ \# \ ws)\ f_n, \quad f_n\ w_n = bar1\ (w_n \ \# \ \cdots \ \# \ w_0 \ \# \ ws)$$

then $(w_n \ \# \ \cdots \ \# \ w_0 \ \# \ ws) \in good$. Note that this need not necessarily be the shortest possible *good* sequence. The induction principle for the *bar* predicate is

$vs \in bar \Longrightarrow$
$(\bigwedge ws.\ ws \in good \Longrightarrow P\ ws) \Longrightarrow$
$(\bigwedge ws.\ (\bigwedge w.\ w \ \# \ ws \in bar) \Longrightarrow (\bigwedge w.\ P\ (w \ \# \ ws)) \Longrightarrow P\ ws) \Longrightarrow P\ vs$

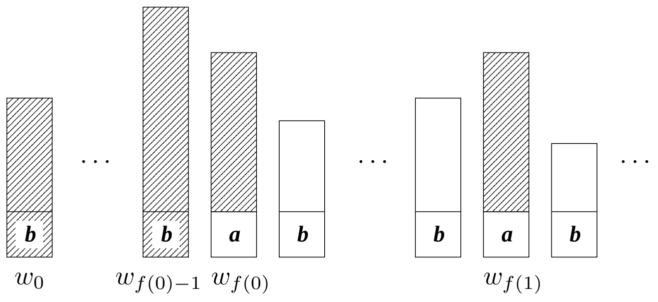


Figure 5.10: Minimal bad sequence argument

It is realized by the recursion combinator

$$\begin{aligned}
&\textit{barT-rec}\ f\ g\ (\textit{bar1 list}) = f\ \textit{list}\\
&\textit{barT-rec}\ f\ g\ (\textit{bar2 list fun}) = g\ \textit{list fun}\ (\lambda x.\ \textit{barT-rec}\ f\ g\ (\textit{fun}\ x))
\end{aligned}$$

The corresponding correctness theorem for this realizer is

$$\begin{aligned}
&(b,\ vs) \in \textit{barR} \Longrightarrow\\
&(\bigwedge ws.\ ws \in \textit{good} \Longrightarrow P\ (f\ ws)\ ws) \Longrightarrow\\
&(\bigwedge ws\ x.\ (\bigwedge w.\ (x\ w,\ w \ \#\ ws) \in \textit{barR}) \Longrightarrow\\
&\qquad\qquad (\bigwedge xa.\ (\bigwedge w.\ P\ (xa\ w)\ (w \ \#\ ws)) \Longrightarrow P\ (g\ ws\ x\ xa)\ ws)) \Longrightarrow\\
&P\ (\textit{barT-rec}\ f\ g\ b)\ vs
\end{aligned}$$

Before explaining the actual proof, we will briefly sketch the main idea of Nash-Williams' classical proof[3], since Coquand's proof can be considered as a constructive version of it. In order to show that every infinite sequence is good, we assume there is a bad sequence and use this to derive a contradiction. If there is a bad sequence, we may also construct a bad sequence $(w_i)_{0\leq i<\omega}$ which is minimal wrt. word length[4]. Since any infinite sequence containing the empty word is necessarily good, each w_i must have the form $a_i \# v_i$. We can find a strictly monotone function f and a letter $a \in \{A, B\}$ such that $a_{f(i)} = a$ for all i. Now consider the sequence $(v_{f(i)})_{0\leq i<\omega}$. If this sequence was bad, we could construct the sequence

$$s \;=\; w_0\ \ldots\ w_{f(0)-1}\ v_{f(0)}\ v_{f(1)}\ \ldots$$

Because the length of $v_{f(0)}$ is smaller than the length of $w_{f(0)}$, and $(w_i)_{0\leq i<\omega}$ is minimal, this sequence must be good. For this to be possible, there must be i and j with $i < f(0)$ and $w_i \trianglelefteq v_{f(j)}$, because both $(w_i)_{0\leq i<\omega}$ and $(v_{f(i)})_{0\leq i<\omega}$ are bad. However, since $v_{f(j)} \trianglelefteq w_{f(j)}$, this implies that $w_i \trianglelefteq w_{f(j)}$, which contradicts the assumption that $(w_i)_{0\leq i<\omega}$ is bad. Hence $(v_{f(i)})_{0\leq i<\omega}$ must be good, which means that there are i and j with $i < j$ and $v_{f(i)} \trianglelefteq v_{f(j)}$, which implies that $a \# v_{f(i)} \trianglelefteq a \# v_{f(j)}$ and therefore $w_{f(i)} \trianglelefteq w_{f(j)}$, which again contradicts the assumption that $(w_i)_{0\leq i<\omega}$ is bad.

To capture the idea underlying the construction of the sequence s shown above, we introduce a relation T, where $(vs,\ ws) \in T\ a$ means that vs is obtained from ws by first copying the prefix of words starting with the letter b, where $a \neq b$, and then appending the tails of words starting with a. This construction principle is illustrated in Figure 5.10, where the shaded

[3]A more general version of this proof for Kruskal's theorem can e.g. be found in the textbook by Baader and Nipkow [8].

[4]A sequence $(w_i)_{0\leq i<\omega}$ is *smaller* than a sequence $(v_i)_{0\leq i<\omega}$ wrt. word length, iff there is a k such that $w_j = v_j$ for all $j < k$ and $\textit{length}(w_k) < \textit{length}(v_k)$.

parts correspond to the sequence s above. In order to define T, we also introduce an auxiliary relation R, where $(vs,\ ws) \in R\ a$ means that ws can be obtained from vs by prefixing each word with the letter a. It should be noted that we could as well have defined $T\ a$ as a function which, given a list ws, yields a list vs. However, we found the relational formulation more convenient to work with.

consts R :: *letter* $\Rightarrow$ (*letter list list* $\times$ *letter list list*) *set*
inductive $R\ a$
intros
  R0: $([],\ []) \in R\ a$
  R1: $(vs,\ ws) \in R\ a \Longrightarrow (w\ \#\ vs,\ (a\ \#\ w)\ \#\ ws) \in R\ a$

consts T :: *letter* $\Rightarrow$ (*letter list list* $\times$ *letter list list*) *set*
inductive $T\ a$
intros
  T0: $a \neq b \Longrightarrow (vs,\ ws) \in R\ b \Longrightarrow (w\ \#\ ws,\ (a\ \#\ w)\ \#\ ws) \in T\ a$
  T1: $(vs,\ ws) \in T\ a \Longrightarrow (v\ \#\ vs,\ (a\ \#\ v)\ \#\ ws) \in T\ a$
  T2: $a \neq b \Longrightarrow (vs,\ ws) \in T\ a \Longrightarrow (vs,\ (b\ \#\ w)\ \#\ ws) \in T\ a$

For example,

$$\begin{aligned}&([w_5,\ w_2,\ B\ \#\ w_1,\ B\ \#\ w_0],\\ &\ [B\ \#\ w_6,\ A\ \#\ w_5,\ B\ \#\ w_4,\ B\ \#\ w_3,\ A\ \#\ w_2,\ B\ \#\ w_1,\ B\ \#\ w_0])\\ &\in T\ A\end{aligned}$$

Before starting with the actual proof, it is useful to prove some lemmas concerning the concepts just introduced. *lemma1* states that each sequence ws containing a word which can be embedded in the word as also contains a word which can be embedded into $a\ \#\ as$. This easily follows from the fact that $as \trianglelefteq a\ \#\ as$. *lemma2* and *lemma3* state that the property *good* is preserved by the relations R and T. Since *good* is defined using L, it is useful to prove similar lemmas for L first. Finally, *lemma4* states that given a list zs of words starting with a, $T\ a$ yields a list ws consisting of the tails of the words in ws.

lemma *lemma1*: $ws \in L\ as \Longrightarrow ws \in L\ (a\ \#\ as)$
lemma *lemma2'*: $(vs,\ ws) \in R\ a \Longrightarrow vs \in L\ as \Longrightarrow ws \in L\ (a\ \#\ as)$
lemma *lemma2*: $(vs,\ ws) \in R\ a \Longrightarrow vs \in good \Longrightarrow ws \in good$
lemma *lemma3'*: $(vs,\ ws) \in T\ a \Longrightarrow vs \in L\ as \Longrightarrow ws \in L\ (a\ \#\ as)$
lemma *lemma3*: $(ws,\ zs) \in T\ a \Longrightarrow ws \in good \Longrightarrow zs \in good$
lemma *lemma4*: $(ws,\ zs) \in R\ a \Longrightarrow ws \neq [] \Longrightarrow (ws,\ zs) \in T\ a$

We will also need the following lemmas about equality on letters:

lemma *letter-neq*: $(a{::}letter) \neq b \Longrightarrow c \neq a \Longrightarrow c = b$
lemma *letter-eq-dec*: $(a{::}letter) = b \vee a \neq b$

Note that *letter-eq-dec* actually yields an algorithm (using case distinctions) for deciding the equality of letters. The actual proof of Higman's lemma is divided into several parts, namely *prop1*, *prop2* and *prop3*. From the computational point of view, these theorems can be thought of as functions transforming trees. Theorem *prop1* states that each sequence ending with the empty word satisfies predicate *bar*, since it can trivially be extended to a *good* sequence by appending any word. This easily follows from the introduction rules for *bar*:

theorem *prop1*: $([]\ \#\ ws) \in bar$ **by** *rules*

The intuition behind *prop2*, which is shown in Figure 5.11, is a bit harder to grasp. Given two trees encoding proofs of $xs \in bar$ and $ys \in bar$, we produce a new tree encoding a proof of $zs \in bar$ by *interleaving* the two input trees. In order to demonstrate that $zs \in bar$, we need to show that, given a sequence of words, we can detect if appending this sequence to zs yields a *good* sequence. This is done by inspecting each word in the sequence to be appended. If the word has the form $a \# w$, we move one step ahead in the tree witnessing $xs \in bar$, whereas we move one step ahead in the tree witnessing $ys \in bar$ if it has the form $b \# w$. Whenever we reach a leaf in one of these trees, we can be sure that, due to the additional constraints on xs, ys and zs, we have turned zs into a good sequence. If the word to be appended is just the empty word [], we know by *prop1* that any following word will make the sequence good. The proof of *prop2* is by double induction on the derivation of $xs \in bar$ and $ys \in bar$ (yielding the induction hypotheses I and I'), followed by a case analysis on the word w to be appended to the sequence zs.

Theorem *prop3* states that we can turn a proof of $xs \in bar$ into a proof of $zs \in bar$, where zs is the list obtained by prefixing each word in the (nonempty) list xs with the letter a. The proof together with its corresponding tree is shown in Figure 5.12. Note that the subtrees of this tree (reachable via edges labelled with words w) are interleavings of other trees formed using *prop2*. In order to prove $zs \in bar$, we again consider all possible words w to be appended to zs. There are essentially two different cases which may occur:

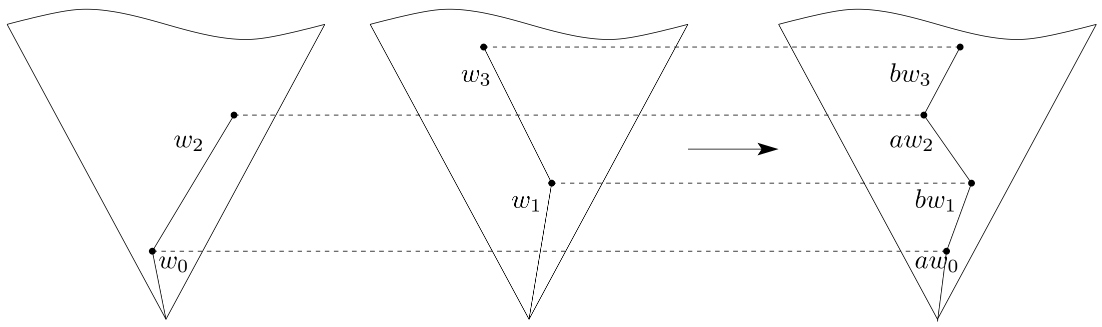
1. If w consists only of b's, i.e. $w = b^n$ for $0 \leq n$, appending words of the form b^n.. or b^m with $m < n$ to the sequence $w \# zs$ will lead to a *good* sequence due to *prop1*, whereas appending words of the form $b^m a$.. with $m < n$ will lead to a *good* sequence due to the fact that $xs \in bar$. The subtrees named *bar* in Figure 5.12 correspond to witnesses of this fact.

2. Similarly, if w contains the letter a, i.e. $w = b^n a$.. with $0 \leq n$, appending words of the form b^n.. to the sequence $w \# zs$ can be shown to lead to a *good* sequence by appealing to the induction hypothesis. Computationally, this corresponds to a recursive call in the function producing the tree, which is why the corresponding subtrees in Figure 5.12 are named *prop3*. Appending words of the form b^m or $b^m a$.. with $m < n$ can be shown to lead to a good sequence by exactly the same argument as in case 1.

The proof of *prop3* is by induction on the derivation of $xs \in bar$, followed by an induction on the word w combined with a case analysis on letters.

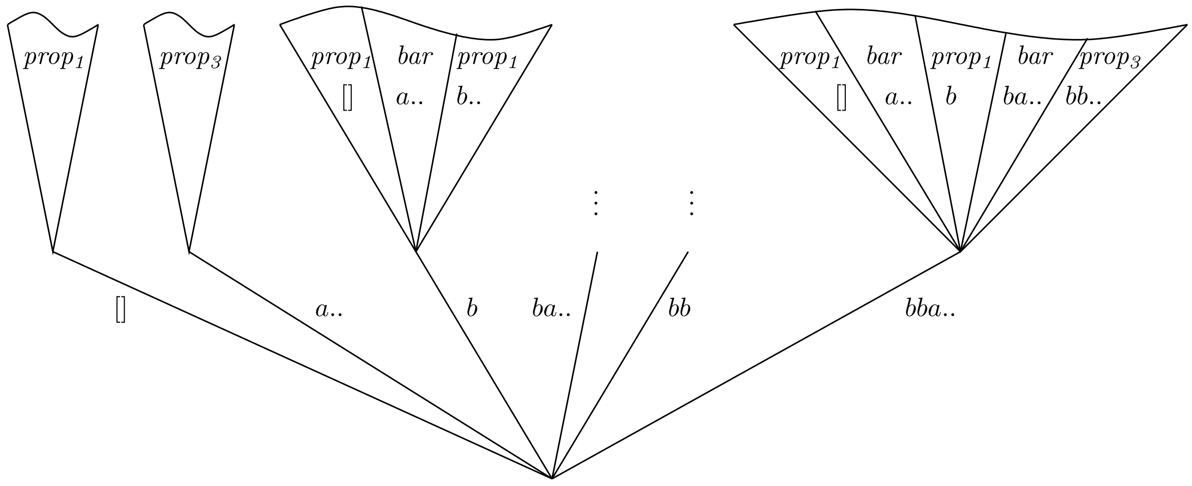
We can now put together the pieces and prove the main theorem. In order to prove that $[] \in bar$, it suffices to show that $[w] \in bar$ for any word w. This can be proved by induction on w. If w is empty, the claim follows by *prop1*. Otherwise, if $w = c \# cs$, we have $[cs] \in bar$ by induction hypothesis, which we can turn into a proof of $[c \# cs] \in bar$ using *prop3*. It should be noted that structural induction on lists can be viewed as the constructive counterpart of the minimality argument used in Nash-Williams' classical proof.

The proof, together with a diagram illustrating the intuition behind it, is shown in Figure 5.13. The shaded parts of the drawing correspond to sequences for which we already know that they are *good* due to the induction hypothesis $[cs] \in bar$. Processing the word w_1 in Figure 5.13 corresponds to following the branch labelled with *bba*.. in Figure 5.12. Processing the word w_2 in Figure 5.13, which starts with at least as many b's as the preceeding word w_1, corresponds to a step in the part of the rightmost subtree in Figure 5.12, which was produced



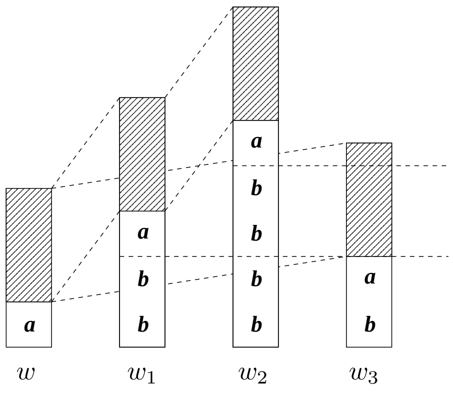
```
theorem prop2:
  assumes ab: a ≠ b and bar: xs ∈ bar
  shows ⋀ys zs. ys ∈ bar ⟹ (xs, zs) ∈ T a ⟹ (ys, zs) ∈ T b ⟹ zs ∈ bar using bar
proof induct
  fix xs zs assume xs ∈ good and (xs, zs) ∈ T a
  show zs ∈ bar by (rule bar1) (rule lemma3)
next
  fix xs ys
  assume I: ⋀w ys zs. ys ∈ bar ⟹ (w # xs, zs) ∈ T a ⟹ (ys, zs) ∈ T b ⟹ zs ∈ bar
  assume ys ∈ bar
  thus ⋀zs. (xs, zs) ∈ T a ⟹ (ys, zs) ∈ T b ⟹ zs ∈ bar
  proof induct
    fix ys zs assume ys ∈ good and (ys, zs) ∈ T b
    show zs ∈ bar by (rule bar1) (rule lemma3)
  next
    fix ys zs assume I': ⋀w zs. (xs, zs) ∈ T a ⟹ (w # ys, zs) ∈ T b ⟹ zs ∈ bar
    and ys: ⋀w. w # ys ∈ bar and Ta: (xs, zs) ∈ T a and Tb: (ys, zs) ∈ T b
    show zs ∈ bar
    proof (rule bar2)
      fix w
      show w # zs ∈ bar
      proof (cases w)
        case Nil
        thus ?thesis by simp (rule prop1)
      next
        case (Cons c cs)
        from letter-eq-dec show ?thesis
        proof
          assume ca: c = a
          from ab have (a # cs) # zs ∈ bar by (rules intro: I ys Ta Tb)
          thus ?thesis by (simp add: Cons ca)
        next
          assume c ≠ a
          with ab have cb: c = b by (rule letter-neq)
          from ab have (b # cs) # zs ∈ bar by (rules intro: I' Ta Tb)
          thus ?thesis by (simp add: Cons cb)
        qed
      qed
    qed
  qed
qed
```

Figure 5.11: Proposition 2



```
theorem prop3:
  assumes bar: xs ∈ bar
  shows ⋀zs. xs ≠ [] ⟹ (xs, zs) ∈ R a ⟹ zs ∈ bar using bar
proof induct
  fix xs zs
  assume xs ∈ good and (xs, zs) ∈ R a
  show zs ∈ bar by (rule bar1) (rule lemma2)
next
  fix xs zs
  assume I: ⋀w zs. w # xs ≠ [] ⟹ (w # xs, zs) ∈ R a ⟹ zs ∈ bar
  and xsb: ⋀w. w # xs ∈ bar and xsn: xs ≠ [] and R: (xs, zs) ∈ R a
  show zs ∈ bar
  proof (rule bar2)
    fix w
    show w # zs ∈ bar
    proof (induct w)
      case Nil
      show ?case by (rule prop1)
    next
      case (Cons c cs)
      from letter-eq-dec show ?case
      proof
        assume c = a
        thus ?thesis by (rules intro: I [simplified] R)
      next
        from R xsn have T: (xs, zs) ∈ T a by (rule lemma4)
        assume c ≠ a
        thus ?thesis by (rules intro: prop2 Cons xsb xsn R T)
      qed
    qed
  qed
qed
```

Figure 5.12: Proposition 3



theorem *higman*: [] ∈ *bar*
proof (*rule bar2*)
  fix *w*
  show [*w*] ∈ *bar*
  proof (*induct w*)
    show [[]] ∈ *bar* **by** (*rule prop1*)
  next
    fix *c cs* **assume** [*cs*] ∈ *bar*
    thus [*c* # *cs*] ∈ *bar* **by** (*rule prop3*) (*simp*, *rules*)
  qed
qed

Figure 5.13: Main theorem

by a recursive call to *prop3*. In contrast, processing the word w_3, which starts with fewer *b*'s than w_1, corresponds to a step in the part of the rightmost subtree labelled with *bar*.

The computational content of the above theorem is an infinitely branching tree, which is a bit difficult to inspect. Using this theorem, we therefore prove an additional statement yielding a program that, given an infinite sequence of words, returns a finite prefix of this sequence which is *good*. Infinite sequences are encoded as functions of type $nat \Rightarrow \alpha$. The fact that a list is a prefix of an infinite sequence can be characterized recursively as follows:

consts *is-prefix* :: *'a list* ⇒ (*nat* ⇒ *'a*) ⇒ *bool*
primrec
  is-prefix [] *f* = *True*
  is-prefix (*x* # *xs*) *f* = (*x* = *f* (*length xs*) ∧ *is-prefix xs f*)

We now prove that an infinite sequence *f* of words has a good prefix *vs*, provided there is a prefix *ws* with *ws* ∈ *bar*. The proof of this theorem, which cannot be found in the paper by Coquand and Fridlender, is by induction on the derivation of *ws* ∈ *bar*. If the derivation tree is a leaf, this means that the current prefix is already good and we simply return it, otherwise we move ahead one step in the tree and continue the search recursively, i.e. apply the induction hypothesis.

theorem *good-prefix-lemma*:
  assumes *bar*: *ws* ∈ *bar*
  shows *is-prefix ws f* ⟹ ∃ *vs*. *is-prefix vs f* ∧ *vs* ∈ *good* **using** *bar*
proof *induct*
  case *bar1*
  thus *?case* **by** *rules*
next
  case (*bar2 ws*)
  have *is-prefix* (*f* (*length ws*) # *ws*) *f* **by** *simp*
  thus *?case* **by** (*rules intro*: *bar2*)
qed

The fact that any infinite sequence has a good prefix can now be obtained as a corollary of this theorem using *higman*:

theorem *good-prefix*: ∃ *vs*. *is-prefix vs f* ∧ *vs* ∈ *good*
  using *higman*
  by (*rule good-prefix-lemma*) *simp*+

As has already been noted, the function extracted from theorem *good-prefix* need not necessarily find the *shortest* good prefix. As an example, consider the following three functions representing sequences of words:

$$f_1(i) = \begin{cases} [A,\ A] & \text{if } i = 0 \\ [B] & \text{if } i = 1 \\ [A,\ B] & \text{if } i = 2 \\ [] & \text{otherwise} \end{cases} \quad f_2(i) = \begin{cases} [A,\ A] & \text{if } i = 0 \\ [B] & \text{if } i = 1 \\ [B,\ A] & \text{if } i = 2 \\ [] & \text{otherwise} \end{cases} \quad f_3(i) = \begin{cases} [A,\ A] & \text{if } i = 0 \\ [B] & \text{if } i = 1 \\ [A,\ B,\ A] & \text{if } i = 2 \\ [] & \text{otherwise} \end{cases}$$

When applied to f_1, *good-prefix* returns the good prefix [[], [], [A, B], [B], [A, A]], which is certainly not the shortest one. The reason for this should become clear when looking at Figure 5.13: In order for the algorithm to recognize that the word [B] can be embedded into some subsequent word, this word has to start with at least one B. However, since the following word starts with an A, the algorithm does not recognize that [B] can be embedded into it. In contrast, when applied to f_2 and f_3, *good-prefix* returns the shortest good prefixes [[B, A], [B], [A, A]] and [[A, B, A], [B], [A, A]], as expected. In the case of f_2, the algorithm recognizes that [B] can be embedded into [B, A], since the latter starts with as many B's as the former. In the case of f_3, the algorithm recognizes that [A] can be embedded into [B, A], and hence, due to lemma *prop3*, also recognizes that [A, A] can be embedded into [A, B, A].

The Isabelle/HOL functions extracted from the proof of theorem *good-prefix* are shown in Figure 5.14. The corresponding ML code, together with auxiliary functions, is presented in Figure 5.15. The correctness theorem for *good-prefix* is

$$\textit{is-prefix}\ (\textit{good-prefix}\ f)\ f \wedge \textit{good-prefix}\ f \in \textit{good}$$

whereas for *higman*, it is simply

$$(\textit{higman},\ []) \in \textit{barR}$$

The correctness theorems for *prop2* and *prop3* are

$$\begin{array}{l} a \neq b \Longrightarrow \\ (\bigwedge x.\ (x,\ xs) \in \textit{barR} \Longrightarrow \\ \quad (\bigwedge xa.\ (xa,\ ys) \in \textit{barR} \Longrightarrow \\ \qquad (xs,\ zs) \in T\ a \Longrightarrow (ys,\ zs) \in T\ b \Longrightarrow (\textit{prop2}\ a\ b\ ys\ zs\ x\ xa,\ zs) \in \textit{barR})) \end{array}$$

and

$$\bigwedge x.\ (x,\ xs) \in \textit{barR} \Longrightarrow xs \neq [] \Longrightarrow (xs,\ zs) \in R\ a \Longrightarrow (\textit{prop3}\ zs\ a\ x,\ zs) \in \textit{barR}$$

To understand the computational behaviour of these functions, it is instructive to derive characteristic equations from their definitions, where the *barT-rec* combinator has been unfolded. For *prop2*, these equations are

```
prop2 a b ys zs (bar1 vs) (bar1 ws) = bar1 zs
prop2 a b ys zs (bar2 vs f) (bar1 ws) = bar1 zs
prop2 a b ys zs (bar1 vs) (bar2 ws g) = bar1 zs
prop2 a b ys zs (bar2 vs f) (bar2 ws g) =
  bar2 zs
   (λus. case us of [] ⇒ prop1 zs
          | c # us ⇒
              case letter-eq-dec c a of
              Left ⇒ prop2 a b ws ((a # us) # zs) (f us) (bar2 ws g)
              | Right ⇒ prop2 a b ws ((b # us) # zs) (bar2 vs f) (g us))
```

```
letter-eq-dec ≡
λx xa.
   case x of A ⇒ case xa of A ⇒ Left | B ⇒ Right
   | B ⇒ case xa of A ⇒ Right | B ⇒ Left

prop1 ≡ λx. bar2 ([] # x) (λw. bar1 (w # [] # x))

prop2 ≡
λx xa xb xc H Ha.
   barT-rec (λws x xa H. bar1 xa)
    (λws xb r xc xd H.
        barT-rec (λws x. bar1 x)
         (λws xb ra xc.
             bar2 xc
              (λw. case w of [] ⇒ prop1 xc
                     | a # list ⇒
                         case letter-eq-dec a x of Left ⇒ r list ws ((x # list) # xc) (bar2 ws xb)
                         | Right ⇒ ra list ((xa # list) # xc)))
         H xd)
    H xb xc Ha

prop3 ≡
λx xa H.
   barT-rec (λws. bar1)
    (λws x r xb.
        bar2 xb
         (list-rec (prop1 xb)
           (λa list H.
               case letter-eq-dec a xa of Left ⇒ r list ((xa # list) # xb)
               | Right ⇒ prop2 a xa ws ((a # list) # xb) H (bar2 ws x))))
    H x

higman ≡ bar2 [] (list-rec (prop1 []) (λa list. prop3 [a # list] a))

good-prefix-lemma ≡ λx. barT-rec (λws. ws) (λws xa r. r (x (length ws)))

good-prefix ≡ λx. good-prefix-lemma x higman
```

Figure 5.14: Program extracted from the proof of Higman's lemma

```
datatype letter = A | B;

datatype nat = id0 | Suc of nat;

datatype barT = bar1 of letter list list | bar2 of letter list list * (letter list -> barT);

fun barT_rec f1 f2 (bar1 list) = f1 list
  | barT_rec f1 f2 (bar2 (list, funa)) = f2 list funa (fn x => barT_rec f1 f2 (funa x));

fun op__43_def0 id0 n = n
  | op__43_def0 (Suc m) n = Suc (op__43_def0 m n);

fun size_def3 [] = id0
  | size_def3 (a :: list) = op__43_def0 (size_def3 list) (Suc id0);

fun good_prefix_lemma x =
  (fn H => barT_rec (fn ws => ws) (fn ws => fn xa => fn r => r (x (size_def3 ws))) H);

fun list_rec f1 f2 [] = f1
  | list_rec f1 f2 (a :: list) = f2 a list (list_rec f1 f2 list);

datatype sumbool = Left | Right;

fun letter_eq_dec x =
  (fn xa =>
    (case x of A => (case xa of A => Left | B => Right)
      | B => (case xa of A => Right | B => Left)));

fun prop1 x = bar2 (([] :: x), (fn w => bar1 (w :: ([] :: x))));

fun prop2 x =
  (fn xa => fn xb => fn xc => fn H => fn Ha =>
    barT_rec (fn ws => fn x => fn xa => fn H => bar1 xa)
      (fn ws => fn xb => fn r => fn xc => fn xd => fn H =>
        barT_rec (fn ws => fn x => bar1 x)
          (fn ws => fn xb => fn ra => fn xc =>
            bar2 (xc, (fn w =>
                        (case w of [] => prop1 xc
                          | (xd :: xe) =>
                              (case letter_eq_dec xd x of
                                Left => r xe ws ((x :: xe) :: xc) (bar2 (ws, xb))
                                | Right => ra xe ((xa :: xe) :: xc))))))
          H xd)
      H xb xc Ha);

fun prop3 x =
  (fn xa => fn H =>
    barT_rec (fn ws => fn x => bar1 x)
      (fn ws => fn x => fn r => fn xb =>
        bar2 (xb, (fn w =>
                    list_rec (prop1 xb)
                      (fn a => fn list => fn H =>
                        (case letter_eq_dec a xa of Left => r list ((xa :: list) :: xb)
                          | Right => prop2 a xa ws ((a :: list) :: xb) H (bar2 (ws, x))))
                      w)))
      H x);

val higman : barT =
  bar2 ([], (fn w =>
             list_rec (prop1 []) (fn a => fn list => fn H => prop3 ((a :: list) :: []) a H) w));

fun good_prefix x = good_prefix_lemma x higman;
```

Figure 5.15: ML code generated from proof of Higman's lemma

whereas *prop3* can be characterized by the equations

```
prop3 xs a (bar1 vs) = bar1 xs
prop3 xs a (bar2 vs f) =
  bar2 xs
   (list-rec (prop1 xs)
     (λb bs H.
         case letter-eq-dec b a of Left ⇒ prop3 ((a # bs) # xs) a (f bs)
         | Right ⇒ prop2 b a vs ((b # bs) # xs) H (bar2 vs f)))
```

Note that of the inductive predicates defined in this section, only *bar* has a computational content. If we were not just interested in a *good* prefix, but also in the exact positions of the two words which can be embedded into each other, we would also have to assign the predicate *good* a computational content.

5.4 Weak normalization for simply-typed Lambda-calculus

As the final and most complex example, we present a fully formalized proof of weak normalization for the simply-typed λ-calculus, i.e. we show that each well-typed term has a β-normal form. One might ask why we do not prove strong normalization in the first place, from which the weak normalization property would then simply follow as a corollary. The reason is that the strong normalization property just abstractly states that each reduction sequence terminates, whereas a proof of weak normalization contains a particular reduction algorithm, which can be uncovered using program extraction. Our formalization is inspired by a paper proof due to Matthes and Joachimski [56]. In contrast to most other proofs to be found in the literature, which are based on the concept of *strong computability* introduced by Tait [115], Matthes' and Joachimski's proof uses a simple inductive characterization of β-normal terms, which turns out to be quite well suited for the purpose of program extraction.

Admittedly, the idea of extracting normalization algorithms from proofs is not completely new. It already dates back to the work of Berger [16], who describes an experiment in extracting a program from a strong normalization proof in the style of Tait, which was formalized using the Minlog theorem prover [15]. As he admits, this proof is not completely formalized inside the theorem prover, since the main induction is done “on the meta level”. Strictly speaking, this proof therefore does not yield a single normalization function, but a whole family of functions. These functions then have to be put together manually, but the resulting program is not typeable in an ML-style type system [16, §3.3]. Also Matthes and Joachimski [56] describe how a program extracted from their proof could look like, but this is only done on paper.

Similar machine-checked formalizations have been carried out by Altenkirch [2, 3], who proved strong normalization for System F using the LEGO proof assistant [63], as well as Barras and Werner [13, 11] who proved decidability of type checking for the Calculus of Constructions and extracted a type checker from this proof using the Coq [12] theorem prover. A formalization of substantial parts of the metatheory of *Pure Type Systems*, also using the LEGO proof assistant, has been done by Pollack [103].

5.4.1 Basic definitions

We start by introducing basic concepts such as terms and substitutions. The following definitions are due to Nipkow [81], who used them as a basis for a proof of the Church-Rosser property for β-reduction. They are reproduced here in order to make the exposition self-contained.

λ-terms are modelled by the datatype *dB* using *de Bruijn indices*, which are encoded by natural numbers.

datatype $dB = Var\ nat \mid App\ dB\ dB \ \mid Abs\ dB$

We use $t \circ u$ as an infix notation for $App\ t\ u$. When substituting a term for a variable inside an abstraction, the indices of all free variables in the term have to be incremented. This is taken care of by the *lift* function

consts $lift :: dB \Rightarrow nat \Rightarrow dB$
primrec
  $lift\ (Var\ i)\ k = (if\ i < k\ then\ Var\ i\ else\ Var\ (i + 1))$
  $lift\ (s \circ t)\ k = lift\ s\ k \circ lift\ t\ k$
  $lift\ (Abs\ s)\ k = Abs\ (lift\ s\ (k + 1))$

Using *lift*, we can now define the substitution $t[u/i]$ of a term u for a variable i in a term t as follows:

consts $subst :: dB \Rightarrow dB \Rightarrow nat \Rightarrow dB$
primrec
  subst-Var: $(Var\ i)[s/k] = (if\ k < i\ then\ Var\ (i - 1)\ else\ if\ i = k\ then\ s\ else\ Var\ i)$
  subst-App: $(t \circ u)[s/k] = t[s/k] \circ u[s/k]$
  subst-Abs: $(Abs\ t)[s/k] = Abs\ (t[lift\ s\ 0\ /\ k{+}1])$

Since the substitution function will be used to specify β-reduction, it actually does not only substitute the term u for the variable i, but also decrements the indices of all other free variables by *1*, to compensate for the disappearance of abstractions during β-reduction. The definition of β-reduction, which is denoted by $s \rightarrow_\beta t$, is as usual:

consts $beta :: (dB \times dB)\ set$
inductive *beta*
intros
  beta: $Abs\ s \circ t \rightarrow_\beta s[t/0]$
  appL: $s \rightarrow_\beta t \Longrightarrow s \circ u \rightarrow_\beta t \circ u$
  appR: $s \rightarrow_\beta t \Longrightarrow u \circ s \rightarrow_\beta u \circ t$
  abs: $s \rightarrow_\beta t \Longrightarrow Abs\ s \rightarrow_\beta Abs\ t$

We also use $\rightarrow_\beta^*$ to denote the transitive closure of $\rightarrow_\beta$. The following congruence rules for $\rightarrow_\beta^*$ are occasionally useful in proofs:

lemma *rtrancl-beta-Abs*: $s \rightarrow_\beta^* s' \Longrightarrow Abs\ s \rightarrow_\beta^* Abs\ s'$
lemma *rtrancl-beta-AppL*: $s \rightarrow_\beta^* s' \Longrightarrow s \circ t \rightarrow_\beta^* s' \circ t$
lemma *rtrancl-beta-AppR*: $t \rightarrow_\beta^* t' \Longrightarrow s \circ t \rightarrow_\beta^* s \circ t'$
lemma *rtrancl-beta-App*: $s \rightarrow_\beta^* s' \Longrightarrow t \rightarrow_\beta^* t' \Longrightarrow s \circ t \rightarrow_\beta^* s' \circ t'$

We will also need the following theorems, asserting that $\rightarrow_\beta$ and $\rightarrow_\beta^*$ are *compatible* with lifting and substitution. The first two of these properties are called *substitutivity* in [56].

theorem *subst-preserves-beta*: $r \rightarrow_\beta s \Longrightarrow (\bigwedge t\ i.\ r[t/i] \rightarrow_\beta s[t/i])$
theorem *subst-preserves-beta'*: $r \rightarrow_\beta^* s \Longrightarrow r[t/i] \rightarrow_\beta^* s[t/i]$
theorem *lift-preserves-beta*: $r \rightarrow_\beta s \Longrightarrow (\bigwedge i.\ lift\ r\ i \rightarrow_\beta lift\ s\ i)$
theorem *lift-preserves-beta'*: $r \rightarrow_\beta^* s \Longrightarrow lift\ r\ i \rightarrow_\beta^* lift\ s\ i$
theorem *subst-preserves-beta2*: $\bigwedge r\ s\ i.\ r \rightarrow_\beta s \Longrightarrow t[r/i] \rightarrow_\beta^* t[s/i]$
theorem *subst-preserves-beta2'*: $r \rightarrow_\beta^* s \Longrightarrow t[r/i] \rightarrow_\beta^* t[s/i]$

In addition to the usual *binary* application operator $s \circ t$, it is often convenient to also have an n-ary application operator $t \circ\circ ts$ for applying a term t to a list of terms ts. To this end, we introduce the abbreviation

translations $t \circ\circ ts \rightleftharpoons \mathit{foldl}\ (op\ \circ)\ t\ ts$

The following equations, describing how lifting and substitution operate on such n-ary applications, are easily established by induction on the list ts:

lemma *lift-map*: $\bigwedge t.\ \mathit{lift}\ (t \circ\circ ts)\ i = \mathit{lift}\ t\ i \circ\circ \mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ i)\ ts$
lemma *subst-map*: $\bigwedge t.\ (t \circ\circ ts)[u/i] = t[u/i] \circ\circ \mathit{map}\ (\lambda t.\ t[u/i])\ ts$

5.4.2 Typed Lambda terms

In this section, we introduce the type system for simply-typed λ-calculus. The typing judgement usually depends on some environment (or context), assigning types to the free variables occurring in a term. Since variables are encoded using de Bruijn indices, it seems convenient to model environments as functions from natural numbers to types. In order to insert a type T into an environment e at a given position i, we define the function

constdefs
  $\mathit{shift} :: (\mathit{nat} \Rightarrow {}'a) \Rightarrow \mathit{nat} \Rightarrow {}'a \Rightarrow \mathit{nat} \Rightarrow {}'a$
  $e\langle i{:}T\rangle \equiv \lambda j.\ \mathit{if}\ j < i\ \mathit{then}\ e\ j\ \mathit{else}\ \mathit{if}\ j = i\ \mathit{then}\ T\ \mathit{else}\ e\ (j - 1)$

where $e\langle i{:}T\rangle$ is syntactic sugar for *shift e i T*. The types of variables with indices less than i are left untouched, whereas the types of variables with indices greater than i are shifted one position up. Instead of working directly with the above definition, we will mainly use the following characteristic theorems for *shift*.

lemma *shift-eq*: $i = j \Longrightarrow (e\langle i{:}T\rangle)\ j = T$
lemma *shift-gt*: $j < i \Longrightarrow (e\langle i{:}T\rangle)\ j = e\ j$
lemma *shift-lt*: $i < j \Longrightarrow (e\langle i{:}T\rangle)\ j = e\ (j - 1)$
lemma *shift-commute*: $e\langle i{:}U\rangle\langle 0{:}T\rangle = e\langle 0{:}T\rangle\langle \mathit{Suc}\ i{:}U\rangle$

Note that the above definition is actually a bit more polymorphic than necessary. We now come to the definition of types. In simply-typed λ-calculus, a type can either be an *atomic* type or a *function* type:

datatype $\mathit{type} = \mathit{Atom}\ \mathit{nat} \mid \mathit{Fun}\ \mathit{type}\ \mathit{type}$

In the sequel, we use $T \Rightarrow U$ as an infix notation for *Fun T U*. In analogy to the concept of an n-ary application, it is also useful to have an n-ary function type operator, which is characterized as follows:

translations $Ts \Rrightarrow T \rightleftharpoons \mathit{foldr}\ \mathit{Fun}\ Ts\ T$

Intuitively, $Ts \Rrightarrow T$ denotes the type of a function whose arguments have the types contained in the list Ts and whose result type is T. The definition of the typing judgement $e \vdash t : T$ is rather straightforward:

inductive *typing*
intros
  Var: $e\ x = T \Longrightarrow e \vdash \mathit{Var}\ x : T$
  Abs: $e\langle 0{:}T\rangle \vdash t : U \Longrightarrow e \vdash \mathit{Abs}\ t : (T \Rightarrow U)$
  App: $e \vdash s : T \Rightarrow U \Longrightarrow e \vdash t : T \Longrightarrow e \vdash (s \circ t) : U$

In the typing rule for abstractions, the argument type T of the function is inserted at position *0* in the environment e when checking the type of the body t. The above typing judgement naturally extends to lists of terms. We write $e \Vdash ts : Ts$ to mean that the terms ts have types Ts. Formally, this extension of the typing judgement to lists of terms is defined as follows:

primrec
$(e \Vdash [] : Ts) = (Ts = [])$
$(e \Vdash (t \# ts) : Ts) =$
$\quad (\mathit{case}\ Ts\ \mathit{of}$
$\qquad [] \Rightarrow \mathit{False}$
$\quad \mid T \# Ts \Rightarrow e \vdash t : T \wedge e \Vdash ts : Ts)$

Using the above typing judgement for lists of terms, we can prove the following elimination and introduction rules for types of n-ary applications:

lemma *list-app-typeE*: $e \vdash t \circ\circ ts : T \Longrightarrow (\bigwedge Ts.\ e \vdash t : Ts \Rrightarrow T \Longrightarrow e \Vdash ts : Ts \Longrightarrow P) \Longrightarrow P$
lemma *list-app-typeI*: $\bigwedge t\ T\ Ts.\ e \vdash t : Ts \Rrightarrow T \Longrightarrow e \Vdash ts : Ts \Longrightarrow e \vdash t \circ\circ ts : T$

When looking at the rule *list-app-typeE* from the program extraction point of view, it is important to note that P may only be instantiated with a computationally relevant formula, if also the premise $e \vdash t \circ\circ ts : T$ is computationally relevant. This is due to the fact that for an arbitrary term t, the types Ts of its argument terms ts can usually not be deduced from e, t, ts and T alone, since t could be an abstraction (see also the discussion about normal proofs in §2.4.2 for a related issue). This information therefore has to be obtained from the typing derivation for $t \circ\circ ts$. However, if the head term t is a variable $\mathit{Var}\ i$, which is always the case if $t \circ\circ ts$ is in normal form, we can find out the argument types Ts without having to inspect the typing derivation, since we can first look up the type of $\mathit{Var}\ i$ in the environment e and then obtain Ts by decomposing this type. We therefore prove the following variant of the rule *list-app-typeE* above.

lemma *var-app-typesE*: $e \vdash \mathit{Var}\ i \circ\circ ts : T \Longrightarrow$
$(\bigwedge Ts.\ e \vdash \mathit{Var}\ i : Ts \Rrightarrow T \Longrightarrow e \Vdash ts : Ts \Longrightarrow P) \Longrightarrow P$

The computational part of the proof uses induction on lists and case analysis on types, whereas the elimination rule for the typing judgement is only needed to establish computationally irrelevant statements at the leaves of the proof. The fact that the premise $e \vdash \mathit{Var}\ i \circ\circ ts : T$ of *var-app-typesE* need not have a computational content, even if P is computationally relevant, is crucial for program extraction, since it leads to a much smaller program which does not involve any computations on typing derivations.

Before we come to the *subject reduction* theorem, which is the main result of this section, we need several additional results about lifting and substitution. The first two of these lemmas state that lifting preserves the type of a term:

lemma *lift-type*: $e \vdash t : T \Longrightarrow (\bigwedge i\ U.\ e\langle i{:}U\rangle \vdash \mathit{lift}\ t\ i : T)$
lemma *lift-types*: $\bigwedge Ts.\ e \Vdash ts : Ts \Longrightarrow e\langle i{:}U\rangle \Vdash (\mathit{map}\ (\lambda t.\ \mathit{lift}\ t\ i)\ ts) : Ts$

The first lemma is easily proved by induction on the typing derivation, whereas the second one, which is just a generalization of the first lemma to lists of terms, can be proved by induction on the list ts using the first result. The other two lemmas state that well-typed substitution preserves the type of terms:

lemma *subst-lemma*: $e \vdash t : T \Longrightarrow (\bigwedge e'\ i\ U\ u.\ e' \vdash u : U \Longrightarrow e = e'\langle i{:}U\rangle \Longrightarrow e' \vdash t[u/i] : T)$
lemma *substs-lemma*: $\bigwedge Ts.\ e \vdash u : T \Longrightarrow e\langle i{:}T\rangle \Vdash ts : Ts \Longrightarrow e \Vdash (\mathit{map}\ (\lambda t.\ t[u/i])\ ts) : Ts$

Again, the proof of the first lemma is by induction on the typing derivation, while the second one is proved by induction on *ts*. We are now ready to prove the *subject reduction* property, i.e. that $\rightarrow_\beta$ preserves the type of a term:

lemma *subject-reduction*: $e \vdash t : T \Longrightarrow (\bigwedge t'.\ t \rightarrow_\beta t' \Longrightarrow e \vdash t' : T)$

The proof is by induction on the typing derivation, where the cases for variables and abstractions are fairly trivial. The case dealing with applications $s \circ t$ can be proved using elimination on $s \circ t \rightarrow_\beta t'$ followed by an application of *subst-lemma*. This theorem easily extends to the transitive closure $\rightarrow_\beta{}^*$ of $\rightarrow_\beta$:

theorem *subject-reduction'*: $t \rightarrow_\beta{}^* t' \Longrightarrow e \vdash t : T \Longrightarrow e \vdash t' : T$

5.4.3 Terms in normal form

The definition which is central to the proof of weak normalization is, of course, that of a term in *normal form*. Intuitively, a term is in normal form, if it is either a variable applied to a list of terms in normal form, or an abstraction whose body is a term in normal form. In order to express the fact that all terms in a list are in normal form, it is convenient to have a predicate *listall P xs*, asserting that a predicate P holds for all elements in the list *xs*:

constdefs
  listall :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$
  $listall\ P\ xs \equiv (\forall\, i.\ i < length\ xs \longrightarrow P\ (xs\ !\ i))$

In the above definition, $xs\ !\ i$ denotes the i-th element of the list *xs*. The predicate *listall* enjoys the following characteristic properties, which will be useful in subsequent proofs:

theorem *listall-nil*: $listall\ P\ []$
theorem *listall-cons*: $P\ x \Longrightarrow listall\ P\ xs \Longrightarrow listall\ P\ (x\ \#\ xs)$
lemma *listall-conj1*: $listall\ (\lambda x.\ P\ x \wedge Q\ x)\ xs \Longrightarrow listall\ P\ xs$
lemma *listall-conj2*: $listall\ (\lambda x.\ P\ x \wedge Q\ x)\ xs \Longrightarrow listall\ Q\ xs$

With the help of *listall*, the set *NF* of terms in normal form can be defined inductively as follows:

consts *NF* :: *dB set*
inductive *NF*
intros
  App: $listall\ (\lambda t.\ t \in NF)\ ts \Longrightarrow Var\ x \circ\circ\ ts \in NF$
  Abs: $t \in NF \Longrightarrow Abs\ t \in NF$

We conclude this section by proving some properties of *NF*, which will be of particular importance for the main proof presented in the next section. As a trivial consequence of the above definition of normal forms, a term consisting of just a variable is in normal form.

lemma *Var-NF*: $Var\ n \in NF$

By substituting a variable i for a variable j in a normal term t, we obtain a term which is still in normal form:

lemma *subst-Var-NF*: $t \in NF \Longrightarrow (\bigwedge i\ j.\ t[Var\ i/j] \in NF)$

The above lemma is easily proved by induction on the derivation of $t \in NF$. If t is in normal form, the term $t \circ Var\ i$ possesses a normal form, too:

lemma *app-Var-NF*: $t \in NF \Longrightarrow \exists\, t'.\ t \circ Var\ i \rightarrow_\beta{}^* t' \wedge t' \in NF$

Again, this result can be proved by induction on the derivation of $t \in NF$, using the previous lemma *subst-Var-NF* in the abstraction case. Finally, lifting a normal term t again yields a normal term:

lemma *lift-NF*: $t \in NF \Longrightarrow (\bigwedge i.\ lift\ t\ i \in NF)$

As usual, the proof is by induction on the derivation of $t \in NF$.

5.4.4 Main theorems

We are now just one step away from our main result, the weak normalization theorem. Actually, the main difficulty is to prove a central lemma, from which weak normalization then follows by a relatively simple argument. The essence of this lemma can be summarized by the slogan "*well-typed substitution preserves the existence of normal forms*". More formally, if we have a well-typed term t in normal form containing a variable i of type U, then the term $t[u/i]$ obtained by substituting a term u of type U for the variable i can be reduced to a normal form t'. The proof of this statement, which we will now discuss in detail, is by main induction on the type U, followed by a side induction on the derivation of $t \in NF$. An interesting point to note is that the induction on the type U will not be performed using the standard structural induction rule

$$(\bigwedge a.\ P\ (Atom\ a)) \Longrightarrow (\bigwedge T1\ T2.\ P\ T1 \Longrightarrow P\ T2 \Longrightarrow P\ (T1 \Rightarrow T2)) \Longrightarrow P\ T$$

for the datatype of types, but using the rule

$$\begin{array}{l}(\bigwedge T.\ (\bigwedge T1\ T2.\ T = T1 \Rightarrow T2 \Longrightarrow P\ T1) \Longrightarrow \\ \qquad (\bigwedge T1\ T2.\ T = T1 \Rightarrow T2 \Longrightarrow P\ T2) \Longrightarrow P\ T) \Longrightarrow \\ P\ T\end{array}$$

which is easily derived from the standard one. The advantage of the latter rule over the standard induction rule is that the application of the induction hypotheses and the case analysis on T are decoupled. This is crucial since the actual structure of the type will be of importance only for one case of the proof, in which the main induction hypothesis is used, whereas in all the other cases, which are proved using the side induction hypothesis, the structure of the type is immaterial.

lemma *subst-type-NF*:
$\bigwedge t\ e\ T\ u\ i.\ t \in NF \Longrightarrow e\langle i{:}U\rangle \vdash t : T \Longrightarrow u \in NF \Longrightarrow e \vdash u : U \Longrightarrow \exists\, t'.\ t[u/i] \rightarrow_\beta{}^* t' \wedge t' \in NF$
(**is** *PROP ?P U* **is** $\bigwedge t\ e\ T\ u\ i.$ *-* $\Longrightarrow$ *PROP ?Q t e T u i U*)

To make the presentation more compact, we use *?P* to abbreviate the main induction hypothesis, and *?Q* to abbreviate the side induction hypothesis, i.e. the above formula without the premise $t \in NF$. We start the proof by performing induction on the type U.

proof (*induct U*)
fix *T t*

For technical reasons, we will also need the following variant of the side induction hypothesis *?Q*, where meta-level implications $\Longrightarrow$ have been replaced by object-level implications $\longrightarrow$:

let $?R = \lambda t.\ \forall\, e\ T'\ u\ i.$
$e\langle i{:}T\rangle \vdash t : T' \longrightarrow u \in NF \longrightarrow e \vdash u : T \longrightarrow (\exists\, t'.\ t[u/i] \rightarrow_\beta{}^* t' \wedge t' \in NF)$

Provided that T is a function type, we may use the main induction hypothesis for both the argument and the result type.

```
assume MI1: ⋀T1 T2. T = T1 ⇒ T2 ⟹ PROP ?P T1
assume MI2: ⋀T1 T2. T = T1 ⇒ T2 ⟹ PROP ?P T2
```

We proceed by side induction on the derivation of $t \in NF$:

```
assume t ∈ NF
thus ⋀e T′ u i. PROP ?Q t e T′ u i T
proof induct
  fix e T′ u i assume uNF: u ∈ NF and uT: e ⊢ u : T
  {
    case (App ts x e- T′- u- i-)
    assume appT: e⟨i:T⟩ ⊢ Var x °° ts : T′
```

In the application case, we have to distinguish whether or not the variable x in the head of the term coincides with the variable i to be substituted.

```
    from nat-eq-dec show ∃t′. (Var x °° ts)[u/i] →β* t′ ∧ t′ ∈ NF
    proof
      assume eq: x = i
      show ?thesis
```

In this case, we do a case analysis on the argument list *ts*. If the argument list is empty, the claim follows trivially.

```
      proof (cases ts)
        case Nil
        with eq have (Var x °° [])[u/i] →β* u by simp
        with Nil and uNF show ?thesis by simp rules
      next
        case (Cons a as)
```

The most difficult case of the proof is the one where the argument list is nonempty, i.e. the term on which the substitution has to be performed has the form $Var\ x\ ^{\circ\circ}\ (a\ \#\ as)$. We start by establishing several typing properties:

```
        with appT have e⟨i:T⟩ ⊢ Var x °° (a # as) : T′ by simp
        then obtain Us
            where varT′: e⟨i:T⟩ ⊢ Var x : Us ⇛ T′ and argsT′: e⟨i:T⟩ ⊩ a # as : Us
          by (rule var-app-typesE)
        from argsT′ obtain T″ Ts where Us: Us = T″ # Ts
          by (cases Us) (rule FalseE, simp+)
        from varT′ and Us have varT: e⟨i:T⟩ ⊢ Var x : T″ ⇒ Ts ⇛ T′ by simp
        from varT eq have T: T = T″ ⇒ Ts ⇛ T′ by cases auto
        with uT have uT′: e ⊢ u : T″ ⇒ Ts ⇛ T′ by simp
        from argsT′ and Us have argsT: e⟨i:T⟩ ⊩ as : Ts by simp
        from argsT′ and Us have argT: e⟨i:T⟩ ⊢ a : T″ by simp
        from argT uT refl have aT: e ⊢ a[u/i] : T″ by (rule subst-lemma)
```

As already noted in §5.4.2, the argument types *Us* can be computed without inspecting the typing derivation. Substitution and normalization will now be performed in several steps. As a first step, we apply substitution and normalization to the tail *as* of the argument list. To this end, we prove the following intermediate statement:

```
        have as: ⋀Us. e⟨i:T⟩ ⊩ as : Us ⟹ listall ?R as ⟹
          ∃as′. Var 0 °° map (λt. lift (t[u/i]) 0) as →β* Var 0 °° map (λt. lift t 0) as′ ∧
            Var 0 °° map (λt. lift t 0) as′ ∈ NF
          (is ⋀Us. - ⟹ - ⟹ ∃as′. ?ex Us as as′)
```

The desired normal forms are guaranteed to exist due to the side induction hypothesis *listall ?R as*. Since we later on want to substitute another term for the head variable *Var 0*, we also have to lift the argument terms, in order to avoid that they are affected by the substitution. In other words, the head variable has to be *new*. The above statement is proved by "reverse induction" on *as*, i.e. elements are appended to the right of the list in the induction step. From the computational point of view, the existentially quantified variable *as′* acts as a kind of *accumulator* for the normalized terms. To save space, the body of the existential quantifier is abbreviated by *?ex*. As expected, the base case for the empty argument list is trivial.

```
proof (induct as rule: rev-induct)
  case (Nil Us)
  with Var-NF have ?ex Us [] [] by simp
  thus ?case ..
next
  case (snoc b bs Us)
```

In the step case, we need to perform substitution and normalization on the argument list *bs* @ [*b*]. By the "reverse induction" hypothesis, we already know the result for the argument list *bs*, whereas the result of applying substitution and normalization to *b* can be computed by appeal to the side induction hypothesis *?R b*. We can then put together the normalized terms using the fact that $\rightarrow_\beta{}^*$ is a congruence wrt. application and the fact that $\rightarrow_\beta{}^*$ is compatible with lifting.

```
  have e⟨i:T⟩ ⊩ bs @ [b] : Us .
  then obtain Vs W where Us: Us = Vs @ [W]
    and bs: e⟨i:T⟩ ⊩ bs : Vs and bT: e⟨i:T⟩ ⊢ b : W by (rule types-snocE)
  from snoc have listall ?R bs by simp
  with bs have ∃bs′. ?ex Vs bs bs′ by (rule snoc)
  then obtain bs′ where
    bsred: Var 0 °° map (λt. lift (t[u/i]) 0) bs →β* Var 0 °° map (λt. lift t 0) bs′
    and bsNF: Var 0 °° map (λt. lift t 0) bs′ ∈ NF by rules
  from snoc have ?R b by simp
  with bT and uNF and uT have ∃b′. b[u/i] →β* b′ ∧ b′ ∈ NF by rules
  then obtain b′ where bred: b[u/i] →β* b′ and bNF: b′ ∈ NF by rules
  from bsNF have listall (λt. t ∈ NF) (map (λt. lift t 0) bs′) by (rule App-NF-D)
  moreover have lift b′ 0 ∈ NF by (rule lift-NF)
  ultimately have listall (λt. t ∈ NF) (map (λt. lift t 0) (bs′ @ [b′]))
    by simp
  hence Var 0 °° map (λt. lift t 0) (bs′ @ [b′]) ∈ NF by (rule NF.App)
  moreover from bred have lift (b[u/i]) 0 →β* lift b′ 0 by (rule lift-preserves-beta′)
  with bsred have (Var 0 °° map (λt. lift (t[u/i]) 0) bs) ° lift (b[u/i]) 0 →β*
    (Var 0 °° map (λt. lift t 0) bs′) ° lift b′ 0 by (rule rtrancl-beta-App)
  ultimately have ?ex Us (bs @ [b]) (bs′ @ [b′]) by simp
  thus ?case ..
qed
from App and Cons have listall ?R as by simp (rules dest: listall-conj2)
with argsT have ∃as′. ?ex Ts as as′ by (rule as)
then obtain as′ where
  asred: Var 0 °° map (λt. lift (t[u/i]) 0) as →β* Var 0 °° map (λt. lift t 0) as′
  and asNF: Var 0 °° map (λt. lift t 0) as′ ∈ NF by rules
```

By using the side induction hypothesis one more time, we can also apply substitution and normalization to the head *a* of the argument list.

```
from App and Cons have ?R a by simp
with argT and uNF and uT have ∃a′. a[u/i] →β* a′ ∧ a′ ∈ NF by rules
then obtain a′ where ared: a[u/i] →β* a′ and aNF: a′ ∈ NF by rules
```

In order to show that the application of *u* to *a*[*u*/*i*] has a normal form, too, we first note that the term

u applied to a new variable again has a normal form. Since the argument type T'' of u is smaller than the type $T = T'' \Rightarrow Ts \Rrightarrow T'$, we can use the main induction hypothesis, together with the previous result and compatibility of $\rightarrow_\beta{}^*$ with substitution, to show that also $u \circ a[u/i]$ has a normal form.

```
from uNF have lift u 0 ∈ NF by (rule lift-NF)
hence ∃ u'. lift u 0 ° Var 0 →β* u' ∧ u' ∈ NF by (rule app-Var-NF)
then obtain u' where ured: lift u 0 ° Var 0 →β* u' and u'NF: u' ∈ NF by rules
from T and u'NF have ∃ ua. u'[a'/0] →β* ua ∧ ua ∈ NF
proof (rule MI1)
  have e⟨0:T''⟩ ⊢ lift u 0 ° Var 0 : Ts ⇛ T'
  proof (rule typing.App)
    from uT' show e⟨0:T''⟩ ⊢ lift u 0 : T'' ⇒ Ts ⇛ T' by (rule lift-type)
    show e⟨0:T''⟩ ⊢ Var 0 : T'' by (rule typing.Var) simp
  qed
  with ured show e⟨0:T''⟩ ⊢ u' : Ts ⇛ T' by (rule subject-reduction')
  from ared aT show e ⊢ a' : T'' by (rule subject-reduction')
qed
then obtain ua where uared: u'[a'/0] →β* ua and uaNF: ua ∈ NF by rules
from ared have (lift u 0 ° Var 0)[a[u/i]/0] →β* (lift u 0 ° Var 0)[a'/0]
  by (rule subst-preserves-beta2')
also from ured have (lift u 0 ° Var 0)[a'/0] →β* u'[a'/0]
  by (rule subst-preserves-beta')
also note uared
finally have (lift u 0 ° Var 0)[a[u/i]/0] →β* ua .
hence uared': u ° a[u/i] →β* ua by simp
```

Finally, since the type $Ts \Rrightarrow T'$ of $u \circ a[u/i]$ is also smaller than the type $T = T'' \Rightarrow Ts \Rrightarrow T'$, we may again use the main induction hypothesis, together with the previous result, the above intermediate statement concerning the application of substitution and normalization to the argument list *as*, as well as compatibility of $\rightarrow_\beta{}^*$ with substitution, to show that also $u \circ a[u/i] \circ\circ \mathit{map}\ (\lambda t.\ t[u/i])\ \mathit{as}$ has a normal form.

```
from T have ∃ r. (Var 0 °° map (λt. lift t 0) as')[ua/0] →β* r ∧ r ∈ NF
proof (rule MI2)
  have e⟨0:Ts ⇛ T'⟩ ⊢ Var 0 °° map (λt. lift (t[u/i]) 0) as : T'
  proof (rule list-app-typeI)
    show e⟨0:Ts ⇛ T'⟩ ⊢ Var 0 : Ts ⇛ T' by (rule typing.Var) simp
    from uT argsT have e ⊩ map (λt. t[u/i]) as : Ts
      by (rule substs-lemma)
    hence e⟨0:Ts ⇛ T'⟩ ⊩ map (λt. lift t 0) (map (λt. t[u/i]) as) : Ts
      by (rule lift-types)
    thus e⟨0:Ts ⇛ T'⟩ ⊩ map (λt. lift (t[u/i]) 0) as : Ts
      by (simp-all add: map-compose [symmetric] o-def)
  qed
  with asred show e⟨0:Ts ⇛ T'⟩ ⊢ Var 0 °° map (λt. lift t 0) as' : T'
    by (rule subject-reduction')
  from argT uT refl have e ⊢ a[u/i] : T'' by (rule subst-lemma)
  with uT' have e ⊢ u ° a[u/i] : Ts ⇛ T' by (rule typing.App)
  with uared' show e ⊢ ua : Ts ⇛ T' by (rule subject-reduction')
qed
then obtain r where rred: (Var 0 °° map (λt. lift t 0) as')[ua/0] →β* r
  and rnf: r ∈ NF by rules
from asred have (Var 0 °° map (λt. lift (t[u/i]) 0) as)[u ° a[u/i]/0] →β*
  (Var 0 °° map (λt. lift t 0) as')[u ° a[u/i]/0]
  by (rule subst-preserves-beta')
also from uared' have (Var 0 °° map (λt. lift t 0) as')[u ° a[u/i]/0] →β*
```

```
      (Var 0 °° map (λt. lift t 0) as')[ua/0] by (rule subst-preserves-beta2')
    also note rred
    finally have (Var 0 °° map (λt. lift (t[u/i]) 0) as)[u ° a[u/i]/0] →β* r .
    with rnf Cons eq show ?thesis
      by (simp add: map-compose [symmetric] o-def) rules
  qed
```

This concludes the proof for the case where $x = i$.

```
next
  assume neq: x ≠ i
  show ?thesis
```

The proof for this case is much easier than the previous one, although it is not, as claimed by Matthes and Joachimski [56, §2.3], completely "trivial". As in the previous case, the side induction hypothesis has to be applied to all terms in the argument list *ts*, which is accomplished by proving an intermediate statement using "reverse induction" on *ts*. Again, the fact that $\rightarrow_\beta{}^*$ is a congruence wrt. application is required in the step case, and the existentially quantified variable *ts'* acts as an accumulator for normalized terms. This time, no lifting is involved, but the head variable may be decremented as a side effect of substitution (see §5.4.1) if $i < x$, which is why we prove the statement for all head variables x' first and perform a case distinction later on.

```
  proof -
    from appT obtain Us
        where varT: e⟨i:T⟩ ⊢ Var x : Us ⇛ T'
        and argsT: e⟨i:T⟩ ⊩ ts : Us
      by (rule var-app-typesE)
    have ts: ⋀Us. e⟨i:T⟩ ⊩ ts : Us ⟹ listall ?R ts ⟹
      ∃ts'. ∀x'. Var x' °° map (λt. t[u/i]) ts →β* Var x' °° ts' ∧ Var x' °° ts' ∈ NF
      (is ⋀Us. - ⟹ - ⟹ ∃ts'. ?ex Us ts ts')
    proof (induct ts rule: rev-induct)
      case (Nil Us)
      with Var-NF have ?ex Us [] [] by simp
      thus ?case ..
    next
      case (snoc b bs Us)
      have e⟨i:T⟩ ⊩ bs @ [b] : Us .
      then obtain Vs W where Us: Us = Vs @ [W]
        and bs: e⟨i:T⟩ ⊩ bs : Vs and bT: e⟨i:T⟩ ⊢ b : W by (rule types-snocE)
      from snoc have listall ?R bs by simp
      with bs have ∃bs'. ?ex Vs bs bs' by (rule snoc)
      then obtain bs' where
        bsred: ⋀x'. Var x' °° map (λt. t[u/i]) bs →β* Var x' °° bs'
        and bsNF: ⋀x'. Var x' °° bs' ∈ NF by rules
      from snoc have ?R b by simp
      with bT and uNF and uT have ∃b'. b[u/i] →β* b' ∧ b' ∈ NF by rules
      then obtain b' where bred: b[u/i] →β* b' and bNF: b' ∈ NF by rules
      from bsred bred have ⋀x'. (Var x' °° map (λt. t[u/i]) bs) ° b[u/i] →β*
        (Var x' °° bs') ° b' by (rule rtrancl-beta-App)
      moreover from bsNF [of 0] have listall (λt. t ∈ NF) bs' by (rule App-NF-D)
      with bNF have listall (λt. t ∈ NF) (bs' @ [b']) by simp
      hence ⋀x'. Var x' °° (bs' @ [b']) ∈ NF by (rule NF.App)
      ultimately have ?ex Us (bs @ [b]) (bs' @ [b']) by simp
      thus ?case ..
    qed
    from App have listall ?R ts by (rules dest: listall-conj2)
    with argsT have ∃ts'. ?ex Ts ts ts' by (rule ts)
```

```
      then obtain $ts'$ where $NF$: $?ex\ Ts\ ts\ ts'$ ..
      from $nat\text{-}le\text{-}dec$ show $?thesis$
      proof
        assume $i < x$
        with $NF$ show $?thesis$ by $simp\ rules$
      next
        assume $\neg\ (i < x)$
        with $NF\ neq$ show $?thesis$ by $(simp\ add\text{: } subst\text{-}Var)\ rules$
      qed
    qed
  qed
```

This concludes the proof for the application case. The abstraction case follows by an easy application of the side induction hypothesis, using the fact that $\rightarrow_\beta{}^*$ is a congruence wrt. abstraction.

```
    next
      case $(Abs\ r\ e\text{-}\ T'\text{-}\ u\text{-}\ i\text{-})$
      assume $absT$: $e\langle i{:}T\rangle \vdash Abs\ r : T'$
      then obtain $R\ S$ where $e\langle 0{:}R\rangle\langle Suc\ i{:}T\rangle \vdash r : S$ by $(rule\ abs\text{-}typeE)\ simp$
      moreover have $lift\ u\ 0 \in NF$ by $(rule\ lift\text{-}NF)$
      moreover have $e\langle 0{:}R\rangle \vdash lift\ u\ 0 : T$ by $(rule\ lift\text{-}type)$
      ultimately have $\exists t'.\ r[lift\ u\ 0/Suc\ i] \rightarrow_\beta{}^* t' \wedge t' \in NF$ by $(rule\ Abs)$
      thus $\exists t'.\ Abs\ r[u/i] \rightarrow_\beta{}^* t' \wedge t' \in NF$
        by $simp\ (rules\ intro\text{: } rtrancl\text{-}beta\text{-}Abs\ NF.Abs)$
    }
  qed
qed
```

Before we can embark on the proof of the main theorem of this section, stating that each well-typed λ-term has a normal form, there is another problem to solve. As has already been discussed before, all the typing information required in the proof of the central lemma *subst-type-NF* was easy to reconstruct even without inspecting the typing derivation, since the terms supplied as an input to the algorithm underlying the proof were already in normal form. This is no longer the case for the main theorem, of course, since its very purpose is the normalization of terms. As these terms do not contain any typing information themselves, this information has to be obtained from the typing derivation. In order be able to formalize the main theorem, we therefore define a computationally relevant copy $e \vdash_R t : T$ of the typing judgement $e \vdash t : T$, where the subscript R stands for *R*elevant. The introduction rules characterizing this judgement are the same as for the original one. In order to plug the previous lemma into the proof of the main theorem, we will need the following rule, stating that the computationally relevant typing judgement implies the computationally irrelevant one:

lemma *rtyping-imp-typing*: $e \vdash_R t : T \Longrightarrow e \vdash t : T$

This rule is easily proved by induction on $e \vdash_R t : T$. Note that the other direction would be provable as well, although, from the program extraction point of view, this would not make much sense, since there cannot be a program corresponding to the proof of a computationally relevant statement by induction on a computationally irrelevant statement.

We are now ready to prove weak normalization, which will be done by induction on the typing derivation $e \vdash_R t : T$. All cases except for the application case are trivial. In order to normalize a term of the form $s \circ t$, we first use the induction hypothesis to compute the normal forms s' and t' of s and t, respectively. To show that also $s' \circ t'$ has a normal form, we first note that the application of a new variable to the term t' is in normal form, so the term obtained by substituting the term s' for this variable has a normal form according to lemma *subst-type-NF*. By transitivity, we can then put together the reduction sequences found in this way, to yield a normal form of $s \circ t$.

```
theorem type-NF: assumes T: $e \vdash_R t : T$
  shows $\exists t'.\ t \rightarrow_\beta{}^* t' \wedge t' \in NF$ using T
proof induct
  case Var
  show ?case by (rules intro: Var-NF)
next
  case Abs
  thus ?case by (rules intro: rtrancl-beta-Abs NF.Abs)
next
  case (App T U e s t)
  from App obtain $s'\ t'$ where
    sred: $s \rightarrow_\beta{}^* s'$ and sNF: $s' \in NF$
    and tred: $t \rightarrow_\beta{}^* t'$ and tNF: $t' \in NF$ by rules
  have $\exists u.\ (Var\ 0 \circ lift\ t'\ 0)[s'/0] \rightarrow_\beta{}^* u \wedge u \in NF$
  proof (rule subst-type-NF)
    have $lift\ t'\ 0 \in NF$ by (rule lift-NF)
    hence $listall\ (\lambda t.\ t \in NF)\ [lift\ t'\ 0]$ by (rule listall-cons) (rule listall-nil)
    hence $Var\ 0 \circ\circ [lift\ t'\ 0] \in NF$ by (rule NF.App)
    thus $Var\ 0 \circ lift\ t'\ 0 \in NF$ by simp
    show $e\langle 0{:}T \Rightarrow U\rangle \vdash Var\ 0 \circ lift\ t'\ 0 : U$
    proof (rule typing.App)
      show $e\langle 0{:}T \Rightarrow U\rangle \vdash Var\ 0 : T \Rightarrow U$
        by (rule typing.Var) simp
      from tred have $e \vdash t' : T$
        by (rule subject-reduction') (rule rtyping-imp-typing)
      thus $e\langle 0{:}T \Rightarrow U\rangle \vdash lift\ t'\ 0 : T$
        by (rule lift-type)
    qed
    from sred show $e \vdash s' : T \Rightarrow U$
      by (rule subject-reduction') (rule rtyping-imp-typing)
  qed
  then obtain u where ured: $s' \circ t' \rightarrow_\beta{}^* u$ and unf: $u \in NF$ by simp rules
  from sred tred have $s \circ t \rightarrow_\beta{}^* s' \circ t'$ by (rule rtrancl-beta-App)
  hence $s \circ t \rightarrow_\beta{}^* u$ using ured by (rule rtrancl-trans)
  with unf show ?case by rules
qed
```

5.4.5 Extracted programs

We conclude this case study with an analysis of the programs extracted from the proofs presented in the previous section. The program corresponding to the proof of the central lemma, which performs substitution and normalization, is shown in Figure 5.16. The outer structure of this program consists of two nested recursion combinators *type-induct-P* and *NFT-rec* corresponding to induction on types and the derivation of normal forms, respectively. The datatype representing the computational content of the inductive definition of normal forms is

datatype $NFT\ =\ Dummy\ \mid\ App\ (dB\ list)\ nat\ (nat \Rightarrow NFT)\ \mid\ Abs\ dB\ NFT$

```
subst-type-NF ≡
λx xa xb xc xd xe H Ha.
  type-induct-P xc
   (λx H2 H2a xa xb xc xd xe H.
       NFT-rec arbitrary
        (λts xa xaa r xb xc xd xe H.
            case nat-eq-dec xa xe of
            Left ⇒ case ts of [] ⇒ (xd, H)
                    | a # list ⇒
                        var-app-typesE-P (xb⟨xe:x⟩) xa (a # list)
                         (λUs. case Us of [] ⇒ arbitrary
                                | T'' # Ts ⇒
                                    case rev-induct-P list (λx H. ([], Var-NF 0))
                                           (λx xa H2 xc Ha.
                                               types-snocE-P xa x xc
                                                (λVs W.
case H2 Vs (fst (fst (listall-snoc-P xa) Ha)) of
(x, y) ⇒
  case snd (fst (listall-snoc-P xa) Ha) xb W xd xe H of
  (xa, ya) ⇒
    (x @ [xa],
     NFT.App (map (λt. lift t 0) (x @ [xa])) 0
      (λxa. snd (listall-snoc-P (map (λt. lift t 0) x)) (App-NF-D y, lift-NF 0 ya) xa))))
                                           Ts (listall-conj2-P-Q list
                                                 (λi. (xaa (Suc i), r (Suc i)))) of
                                    (x, y) ⇒
                                      case snd (xaa 0, r 0) xb T'' xd xe H of
                                      (xa, ya) ⇒
                                        case app-Var-NF 0 (lift-NF 0 H) of
                                        (xd, yb) ⇒
                                          case H2 T'' (Ts ⇛ xc) xd xb (Ts ⇛ xc) xa 0 yb ya of
                                          (xa, ya) ⇒
                                            case H2a T'' (Ts ⇛ xc)
(foldl dB.App (dB.Var 0) (map (λt. lift t 0) x)) xb xc xa 0 y ya of
                                            (x, y) ⇒ (x, y))
            | Right ⇒
                var-app-typesE-P (xb⟨xe:x⟩) xa ts
                 (λUs. case rev-induct-P ts (λx H. ([], λx. Var-NF x))
                               (λx xa H2 xc Ha.
                                   types-snocE-P xa x xc
                                    (λVs W. case H2 Vs (fst (fst (listall-snoc-P xa) Ha)) of
                                               (x, y) ⇒
                                                 case snd (fst (listall-snoc-P xa) Ha) xb W xd xe H of
                                                 (xa, ya) ⇒
                                                   (x @ [xa],
λxb. NFT.App (x @ [xa]) xb (snd (listall-snoc-P x) (App-NF-D (y 0), ya)))))
                               Us (listall-conj2-P-Q ts (λz. (xaa z, r z))) of
                        (x, y) ⇒
                          case nat-le-dec xe xa of
                          Left ⇒ (foldl (λu ua. dB.App u ua) (dB.Var (xa − Suc 0)) x,
                                    y (xa − Suc 0))
                          | Right ⇒ (foldl (λu ua. dB.App u ua) (dB.Var xa) x, y xa)))
        (λt x r xa xb xc xd H.
            abs-typeE-P xb
             (λU V. case case r (λu. (xa⟨0:U⟩) u) V (lift xc 0) (Suc xd) (lift-NF 0 H) of
                            (x, y) ⇒ (dB.Abs x, NFT.Abs x y) of
                      (x, y) ⇒ (x, y)))
          H (λu. xb u) xc xd xe)
    x xa xd xe xb H Ha
```

Figure 5.16: Program extracted from *subst-type-NF*

```
subst-Var-NF ≡
λx xa H.
   NFT-rec arbitrary
    (λts x xa r xb xc.
         case nat-eq-dec x xc of
         Left ⇒ NFT.App (map (λt. t[dB.Var xb/xc]) ts) xb
                   (subst-terms-NF ts xb xc (listall-conj1-P-Q ts (λz. (xa z, r z)))
                     (listall-conj2-P-Q ts (λz. (xa z, r z))))
        | Right ⇒
             case nat-le-dec xc x of
             Left ⇒ NFT.App (map (λt. t[dB.Var xb/xc]) ts) (x − Suc 0)
                       (subst-terms-NF ts xb xc (listall-conj1-P-Q ts (λz. (xa z, r z)))
                         (listall-conj2-P-Q ts (λz. (xa z, r z))))
            | Right ⇒
                 NFT.App (map (λt. t[dB.Var xb/xc]) ts) x
                  (subst-terms-NF ts xb xc (listall-conj1-P-Q ts (λz. (xa z, r z)))
                    (listall-conj2-P-Q ts (λz. (xa z, r z)))))
    (λt x r xa xb. NFT.Abs (t[dB.Var (Suc xa)/Suc xb]) (r (Suc xa) (Suc xb))) H x xa

app-Var-NF ≡
λx. NFT-rec arbitrary
      (λts xa xaa r.
           (foldl dB.App (dB.Var xa) (ts @ [dB.Var x]),
            NFT.App (ts @ [dB.Var x]) xa
             (snd (listall-app-P ts)
               (listall-conj1-P-Q ts (λz. (xaa z, r z)),
                listall-cons-P (Var-NF x) listall-nil-eq-P))))
      (λt xa r. (t[dB.Var x/0], subst-Var-NF x 0 xa))

lift-NF ≡
λx H. NFT-rec arbitrary
          (λts x xa r xb.
               case nat-le-dec x xb of
               Left ⇒ NFT.App (map (λt. lift t xb) ts) x
                         (lift-terms-NF ts xb (listall-conj1-P-Q ts (λz. (xa z, r z)))
                           (listall-conj2-P-Q ts (λz. (xa z, r z))))
              | Right ⇒
                   NFT.App (map (λt. lift t xb) ts) (Suc x)
                    (lift-terms-NF ts xb (listall-conj1-P-Q ts (λz. (xa z, r z)))
                      (listall-conj2-P-Q ts (λz. (xa z, r z)))))
          (λt x r xa. NFT.Abs (lift t (Suc xa)) (r (Suc xa))) H x

type-NF ≡
λH. rtypingT-rec (λe x T. (dB.Var x, Var-NF x))
       (λe T t U x r. case r of (x, y) ⇒ (dB.Abs x, NFT.Abs x y))
       (λe s T U t x xa r ra.
            case r of
            (x, y) ⇒
              case ra of
              (xa, ya) ⇒
                case case subst-type-NF (dB.App (dB.Var 0) (lift xa 0)) e 0 (T ⇒ U) U x
                             (NFT.App [lift xa 0] 0 (listall-cons-P (lift-NF 0 ya) listall-nil-P)) y of
                       (x, y) ⇒ (x, y) of
                (x, y) ⇒ (x, y))
       H
```

Figure 5.17: Program extracted from lemmas and main theorem

Note that the universal quantifier in the definition of the predicate *listall*, which is used in the first introduction rule of *NF* shown in §5.4.3, gives rise to the function type $nat \Rightarrow NFT$ in the list of argument types for the constructor *App* in the above datatype definition. Since the constructors *App* and *Abs* both refer to the type *NFT* to be defined recursively, another *Dummy* constructor is required in order to ensure non-emptiness of the datatype.

The recursion combinator *NFT-rec* occurring in the program shown in Figure 5.16 has three functions as arguments, corresponding to the constructors of the above datatype. The first function corresponds to the *Dummy* constructor. Since this constructor may never occur, we may supply an *arbitrary* function as an argument, which, when generating executable code, may be implemented by a function raising an exception on invocation. The second function corresponds to the application case of the proof. It contains a case distinction (using function *nat-eq-dec*) on whether the variable *xa* coincides with the variable *xe*[5]. The first case (labelled with *Left*), which is the more difficult one, contains another case distinction on the structure of the argument list. The second case (labelled with *Right*) is the easier one. It contains another case distinction (using function *nat-le-dec*) on whether $xe < xa$. In the "*Left*" case, the variable in the head of the term is decremented, whereas it remains unchanged in the "*Right*" case. In both the case for $xa = xe$ and $xa \neq xe$ the function *rev-induct-P* is used to apply the normalization function to a list of terms. The last seven lines of the program shown in Figure 5.16 contain the relatively trivial program corresponding to the proof for the abstraction case. The correctness theorem corresponding to the program *subst-type-NF* is

$$\begin{array}{l}
\bigwedge x.\ (x,\ t) \in NFR \Longrightarrow \\
\quad e\langle i{:}U\rangle \vdash t : T \Longrightarrow \\
\quad (\bigwedge xa.\ (xa,\ u) \in NFR \Longrightarrow \\
\qquad e \vdash u : U \Longrightarrow \\
\qquad t[u/i] \rightarrow_\beta{}^* \mathit{fst}\ (\mathit{subst\text{-}type\text{-}NF}\ t\ e\ i\ U\ T\ u\ x\ xa) \wedge \\
\qquad (\mathit{snd}\ (\mathit{subst\text{-}type\text{-}NF}\ t\ e\ i\ U\ T\ u\ x\ xa),\ \mathit{fst}\ (\mathit{subst\text{-}type\text{-}NF}\ t\ e\ i\ U\ T\ u\ x\ xa)) \in NFR)
\end{array}$$

where *NFR* is the realizability predicate corresponding to the datatype *NFT*, which is inductively defined by the rules

$$\begin{array}{l}
\forall i.\ i < \mathit{length}\ ts \longrightarrow (\mathit{nfs}\ i,\ ts\ !\ i) \in NFR \Longrightarrow \\
(NFT.App\ ts\ x\ \mathit{nfs},\ \mathit{foldl}\ dB.App\ (dB.Var\ x)\ ts) \in NFR \\
(\mathit{nf},\ t) \in NFR \Longrightarrow (NFT.Abs\ t\ \mathit{nf},\ dB.Abs\ t) \in NFR
\end{array}$$

Note that $(\mathit{nf},\ t) \in NFR \Longrightarrow t \in NF$, which is easily proved by induction on the derivation of $(\mathit{nf},\ t) \in NFR$.

The programs corresponding to the main theorem *type-NF*, as well as to some lemmas, are shown in Figure 5.17. The function *type-NF* is defined by recursion on the datatype

$$\begin{array}{l}
\mathsf{datatype}\ \mathit{rtypingT}\ = \\
\qquad Var\ (nat \Rightarrow type)\ nat\ type \\
\quad \mid\ Abs\ (nat \Rightarrow type)\ type\ dB\ type\ \mathit{rtypingT} \\
\quad \mid\ App\ (nat \Rightarrow type)\ dB\ type\ type\ dB\ \mathit{rtypingT}\ \mathit{rtypingT}
\end{array}$$

representing the computational content of the typing derivation. The correctness statement for the main function *type-NF* is

$$\bigwedge x.\ (x,\ e,\ t,\ T) \in \mathit{rtypingR} \Longrightarrow t \rightarrow_\beta{}^* \mathit{fst}\ (\mathit{type\text{-}NF}\ x) \wedge (\mathit{snd}\ (\mathit{type\text{-}NF}\ x),\ \mathit{fst}\ (\mathit{type\text{-}NF}\ x)) \in NFR$$

[5] Due to the numerous transformations, which are performed on the proof before extraction, variable names in the extracted programs may often differ from those in the original Isar proof document.

where the realizability predicate *rtypingR* corresponding to the computationally relevant version of the typing judgement is inductively defined by the rules

$$
\begin{array}{l}
e\ x = T \Longrightarrow (rtypingT.Var\ e\ x\ T,\ e,\ dB.Var\ x,\ T) \in rtypingR \\
(ty,\ e\langle 0{:}T\rangle,\ t,\ U) \in rtypingR \Longrightarrow (rtypingT.Abs\ e\ T\ t\ U\ ty,\ e,\ dB.Abs\ t,\ T \Rightarrow U) \in rtypingR \\
(ty,\ e,\ s,\ T \Rightarrow U) \in rtypingR \Longrightarrow \\
(ty',\ e,\ t,\ T) \in rtypingR \Longrightarrow (rtypingT.App\ e\ s\ T\ U\ t\ ty\ ty',\ e,\ dB.App\ s\ t,\ U) \in rtypingR
\end{array}
$$

The reduction relation $\rightarrow_\beta{}^*$ has been chosen to have no computational content, since we are only interested in the normal form of a term, and not the actual *reduction sequence* leading to it.

Compared to the programs which are extracted "manually" by Matthes and Joachimski [56, §2.3], the automatically extracted programs presented in this section are certainly more complicated and harder to read. This is due to the fact that, although (according to the proof) the main recursion in the program given by Matthes and Joachimski should be over types, no type information is mentioned in the extracted program at all. Moreover, the program looks as if it were defined by recursion over terms, whereas, strictly speaking, it should involve recursion over the datatype *NFT* representing the computational content of the inductive characterization of normal forms.

Due to the considerable size of the proof of lemma *subst-type-NF*, some care was necessary in order to arrive at an extracted program of reasonable size. In a first attempt, we extracted a program of about 600 lines. The reason for this enormous size was the somewhat naive choice of $\lambda p\ pq.\ pq\ (fst\ p)\ (snd\ p)$ as a realizer for the existential elimination rule, which lead to an exponential blowup of the program size due to the double occurrence of the program p in the term. Replacing this realizer by the more efficient $\lambda p\ pq.\ case\ p\ of\ (x,\ y) \Rightarrow pq\ x\ y$ helped to cut down the size of the extracted program to slightly more than 100 lines (see also §4.3.3).

5.5 Discussion

A common prejudice about program extraction is that it is only applicable to toy examples and leads to more complicated formalizations, since everything has to be proved constructively. By the case studies in the previous sections, we have demonstrated that this is not the case, though. In particular, the proof of weak normalization for the simply-typed λ-calculus is a quite large example, consisting of over 500 lines of Isabelle code. It is interesting to note that this proof was obtained by modifying a similar proof of strong normalization, which had been done in Isabelle by the author of this thesis long before the program extraction framework was available. Although this proof was done completely without program extraction in mind, it did not make use of any classical proof principles, so no additional effort was necessary to make it constructive.

A rather impressive example for the power of program extraction has been given by the Foundations of Computer Science group from Nijmegen University. Cruz-Filipe and Spitters [30] have shown how to extract a program from a proof of the *Fundamental Theorem of Algebra* (FTA), which had already been formalized earlier in Coq by Geuvers et al. [41]. One of the lessons learned from this project is that program extraction is not a magic push-button technique, which somehow yields programs for free. Due to the size of the FTA formalization, which consisted of about 930K of Coq code, various optimization techniques had to be applied in order to reduce the size of the extracted program from initially 15Mb to (currently) 3Mb

[30, §4.1]. Developing and investigating such optimization techniques is a promising research area, and a lot of interesting applications are to be expected in the future.

Chapter 6

Executing higher order logic specifications

6.1 Introduction

It has long been recognized by the theorem proving community that the ability to generate executable code from specifications is essential for *validation* purposes. Due to the gap between a "real world" system and its formal model in a theorem prover, trying out prototypes generated from specifications on test cases helps to increase the confidence in the adequacy of the formal model. Practical experience has shown that it can often be advantageous to execute specifications already before one embarks on proving theorems about them. This helps to detect flaws in specifications quite early in the development process, and thus avoids wasting time on trying to carry out proof attempts based on a possibly faulty specification. One of the first theorem provers to take the idea of executable specifications seriously, was the Boyer-Moore system, as well as its successor ACL2 [58]. We have already mentioned earlier that the notion of executability is also central to constructive type theories as implemented for example in Coq [12] or Nuprl [27], which can be viewed as functional programming languages with a very rich type system.

In the previous chapter about program extraction, we have already casually made use of the possibility to generate executable code from definitions in an object logic. So far, these have mainly been definitions of inductive datatypes and primitive recursive functions, whose translation to a functional programming language was fairly straightforward. However, many realistic formalizations, such as specifications of *operational semantics* of programming languages, not only involve recursive functions, but also inductive predicates (or relations), or even a mixture of both. As an example of such a specification, consider again the β-reduction relation presented in the case study about weak normalization from §5.4. While the β-reduction relation itself was defined inductively, it also made use of recursive functions, such as substitution and lifting. In §5.4, our approach of getting from the inductive specification of β-reduction to an executable program was to give a *constructive* proof of the fact that each well-typed λ-term can be reduced to a normal form, and then *extract* a program from this proof which provably satisfies the specification. In this chapter, we will present an alternative approach of giving a computational meaning to inductively defined predicates by directly interpreting them as a *logic program.*

Ordinary logic programming languages such as Prolog suffer from the problem that they do not allow for a smooth combination of predicates and functions. Although functions could be turned into predicates by "flattening" them, this seems somewhat unnatural. In this chapter, we therefore present a method for translating inductively defined relations to functional programs. Instead of *unification*, the translated program will use the built-in *pattern matching* mechanism of functional programming languages. In order to ensure that predicates can be executed in a functional style, we introduce a so-called *mode system*, which captures the direction of *dataflow* in the rules characterizing an inductive relation. Although this translation cannot handle full Prolog, it has turned out to be sufficient for most practical applications. Moreover, it nicely integrates with functional programs, and can also be extended to the higher-order case, e.g. to predicates taking other predicates as arguments.

A problem with higher order logic specifications is that executable and non-executable parts are often not clearly separated. As a first step, we therefore identify an executable subset of higher order logic.

6.2 An executable subset of Isabelle/HOL

As promised in the introduction, we now give a more precise definition of the executable subset of Isabelle/HOL. As a starting point, we briefly review the main ingredients of HOL specifications, which we have already encountered in previous chapters.

inductive datatypes can be defined by specifying their *constructors*, e.g.

datatype $nat \ = \ 0 \mid Suc\ nat$

recursive functions can be defined by specifying several characteristic equations, e.g.

primrec
$add\ 0\ y \ = \ y$
$add\ (Suc\ x)\ y \ = \ Suc\ (add\ x\ y)$

All functions in HOL must be terminating. Supported recursion schemes are *primitive recursion* (primrec) and *well-founded recursion* (recdef) [111].

inductive relations (or predicates) can be defined by specifying their *introduction rules*, e.g.

inductive
$0 \in even$
$x \in even \Longrightarrow Suc\ (Suc\ x) \in even$

Introduction rules are essentially *Horn Clauses*, which are also used in logic programming languages such as Prolog.

Recursive functions and inductive definitions may also be intermixed: For example, an inductive predicate may refer to a recursive function and vice versa.

Executable elements of HOL specifications We now inductively define the elements an *executable* HOL specification may consist of:

- **Executable terms** contain only executable constants
- **Executable constants** can be one of the following
  - executable inductive relations
  - executable recursive functions
  - constructors, recursion and case combinators of executable datatypes
  - operators on executable primitive types such as bool, i.e. the usual propositional operators $\wedge$, $\vee$ and $\neg$, as well as `if _ then _ else _`.
- **Executable datatypes**, where each constructor argument type is again an executable datatype or an executable primitive type such as bool or $\Rightarrow$.
- **Executable inductive relations**, whose introduction rules have the form

  $$(u^1_1,\ \ldots,\ u^1_{n_1}) \in q_1 \Longrightarrow \ldots \Longrightarrow (u^m_1,\ \ldots,\ u^m_{n_m}) \in q_m \Longrightarrow (t_1,\ \ldots,\ t_k) \in p$$

  where u^i_j and t_i are executable terms and q_i is either p or some other executable inductive relation. In addition, also arbitrary executable terms not of the form $(\ldots) \in p_i$, so-called *side conditions*, which may not contain p, are allowed as premises of introduction rules.
- **Executable recursive functions**, i.e. sets of rewrite rules, whose left-hand side contains only constructor patterns with distinct variables, and the right-hand side is an executable term.

In the sequel, we write $\mathcal{C}$ to denote the set of datatype constructors. The non-executable elements of HOL are, among others, arbitrary universal and existential quantification, equality of objects having higher-order types, Hilbert's selection operator ε, arbitrary type definitions (other than datatypes) or inductive definitions whose introduction rules contain quantifiers, like

$$(\bigwedge y.\ (y,\ x) \in r \Longrightarrow y \in acc\ r) \Longrightarrow x \in acc\ r$$

Execution What exactly do we mean by *execution* of specifications? Essentially, execution means finding solutions to queries. A *solution* σ is a mapping of variables to *closed solution terms*. A term t is called a *solution term* iff

- t is of function type, or
- $t = c\ t_1\ \ldots\ t_n$, where the t_i are solution terms and $c \in \mathcal{C}$.

Let solve be a function that returns for each query a set of solutions. We distinguish two kinds of queries:

Functional queries have the form $t = X$, where t is a closed executable term and X is a variable. Queries of this kind should return at most one solution, e.g. $\mathsf{solve}(add\ 0\ (Suc\ 0) = X) = \{[X \mapsto Suc\ 0]\}$

Relational queries have the form $(t_1, \ldots, t_n) \in r$, where r is an executable inductively defined relation and t_i is either a closed executable term or a variable. A query Q of this kind returns a set of solutions $\mathsf{solve}(Q)$. Note that the set returned by solve may also be empty, e.g. $\mathsf{solve}(\mathit{Suc}\ 0 \in \mathit{even}) = \{\}$, or infinite, e.g. $\mathsf{solve}(X \in \mathit{even}) = \{[X \mapsto 0], [X \mapsto \mathit{Suc}\ (\mathit{Suc}\ 0)], \ldots\}$.

All relational queries have to be *well-moded* in order to be executable. We will make this notion more precise in §6.3.1.

It is important to point out the difference between the *execution* of inductively defined predicates as a logic program, and the *extraction* of programs from proofs involving inductive predicates as described in §4.3.5. The latter is concerned with *representing* derivations of statements $(t_1, \ldots, t_n) \in r$ as elements of a datatype, whereas the former corresponds to the *search* for such a derivation.

6.3 Compiling functional logic specifications

Functional-logic programming languages such as Curry [44] should be ideal target languages for code generation from HOL specifications. But although such languages contain many of the required concepts and there is an impressive amount of research in this area, the implementations which are currently available are not always satisfactory. We therefore decided to choose ML, the implementation language of Isabelle, as a target language. Datatypes and recursive functions can be translated to ML in a rather straightforward way, with only minor syntactic modifications. Therefore, this section concentrates on the more interesting task of translating inductive relations to ML. The translation is based on assigning *modes* to relations, a well-known standard technique for the analysis and optimization of logic programs [68]. Mode systems similar to the one described here have also been studied by Stärk [113] and Dubois [35].

6.3.1 Mode analysis

In order to translate a predicate into a function, the direction of *dataflow* has to be analyzed, i.e. it has to be determined which arguments are *input* and which are *output*. Note that for a predicate there may be more than one possible direction of dataflow. For example, the predicate

$$(\mathit{Nil},\ ys,\ ys) \in \mathit{append}$$
$$(xs,\ ys,\ zs) \in \mathit{append} \Longrightarrow (\mathit{Cons}\ x\ xs,\ ys,\ \mathit{Cons}\ x\ zs) \in \mathit{append}$$

may be given two lists $xs = [1,\ 2]$ and $ys = [3,\ 4]$ as input, the output being the list $zs = [1,\ 2,\ 3,\ 4]$. We may as well give a list $zs = [1,\ 2,\ 3,\ 4]$ as an input, the output being a sequence of pairs of lists xs and ys, where zs is the result of appending xs and ys, namely $xs = [1,\ 2,\ 3,\ 4]$ and $ys = []$, or $xs = [1,\ 2,\ 3]$ and $ys = [4]$, or $xs = [1,\ 2]$ and $ys = [3,\ 4]$, etc.

Mode assignment A specific direction of dataflow is called a *mode*. We describe a mode of a predicate by a set of indices, which denote the positions of the input arguments. In the above example, the two modes described were $\{1,\ 2\}$ and $\{3\}$. Given a set of predicates P, a relation *modes* is called a *mode assignment* if

$$\mathit{modes} \subseteq \{(p,\ M) \mid p \in P \wedge M \subseteq \{1,\ \ldots,\ \mathsf{arity}\ p\}\}$$

The set

$$\mathit{modes}\ p = \{M \mid (p,\ M) \in \mathit{modes}\} \qquad \subseteq \mathcal{P}(\{1,\ \ldots,\ \mathsf{arity}\ p\})$$

is the set of modes assigned to predicate p.

Consistency of modes A mode M is called *consistent* with respect to a mode assignment *modes* and a clause

$$(u_1^1,\ \ldots,\ u_{n_1}^1) \in q_1 \Longrightarrow \ldots \Longrightarrow (u_1^m,\ \ldots,\ u_{n_m}^m) \in q_m \Longrightarrow (t_1,\ \ldots,\ t_k) \in p$$

if there exists a permutation π and sets of variable names $v_0, \ldots, v_m$ such that

(1) $v_0 = \mathsf{vars\_of}\ (\mathsf{args\_of}\ M\ (t_1,\ \ldots,\ t_k))$

(2) $\forall 1 \leq i \leq m.\ \exists M' \in \mathit{modes}\ q_{\pi(i)}.\ M' \subseteq \mathsf{known\_args}\ v_{i-1}\ (u_1^{\pi(i)},\ \ldots,\ u_{n_{\pi(i)}}^{\pi(i)})$

(3) $\forall 1 \leq i \leq m.\ v_i = v_{i-1} \cup \mathsf{vars\_of}\ (u_1^{\pi(i)},\ \ldots,\ u_{n_{\pi(i)}}^{\pi(i)})$

(4) $\mathsf{vars\_of}\ (t_1,\ \ldots,\ t_k) \subseteq v_m$

The permutation π denotes a suitable execution order for the predicates $q_1, \ldots, q_m$ in the body of p, where v_i is the set of variables whose value is known after the ith execution step. Condition (1) means that initially, when invoking mode M of predicate p, the values of all variables occurring in the input arguments of the clause head are known. Condition (2) means that in order to invoke a mode M' of a predicate $q_{\pi(i)}$, all of the predicate's input arguments which are specified by M' must be known. According to condition (3), the values of all arguments of $q_{\pi(i)}$ are known after its execution. Finally, condition (4) states that the values of all variables occurring in the clause head of p must be known. Here, function $\mathsf{args\_of}\ M$ returns the tuple of input arguments specified by mode M, e.g.

$$\mathsf{args\_of}\ \{1,\ 2\}\ (\mathit{Cons}\ x\ \mathit{xs},\ \mathit{ys},\ \mathit{Cons}\ x\ \mathit{zs}) = (\mathit{Cons}\ x\ \mathit{xs},\ \mathit{ys})$$

Function $\mathsf{vars\_of}$ returns all variables occurring in a tuple, e.g.

$$\mathsf{vars\_of}\ (\mathit{Cons}\ x\ \mathit{xs},\ \mathit{ys}) = \{x,\ \mathit{xs},\ \mathit{ys}\}$$

Given some set of variables and an argument tuple, $\mathsf{known\_args}$ returns the indices of all arguments, whose value is fully known, provided the values of the variables given are known, e.g.

$$\mathsf{known\_args}\ \{x,\ \mathit{xs},\ \mathit{ys}\}\ (\mathit{Cons}\ x\ \mathit{xs},\ \mathit{ys},\ \mathit{Cons}\ x\ \mathit{zs}) = \{1,\ 2\}$$

Mode inference We write

$$\mathsf{consistent}\ (p,\ M)\ \mathit{modes}$$

if mode M of predicate p is consistent with respect to all clauses of p, under the mode assignment *modes*. Let

$$\Gamma(\mathit{modes}) = \{(p,\ M) \mid (p,\ M) \in \mathit{modes} \wedge \mathsf{consistent}\ (p,\ M)\ \mathit{modes}\}$$

Then the greatest set of allowable modes for a set of predicates P is the greatest fixpoint of Γ. According to Kleene's fixpoint theorem, since Γ is monotone and its domain is finite, this fixpoint can be obtained by finite iteration: we successively apply Γ, starting from the greatest mode assignment

$$\{(p,\ M) \mid p \in P \wedge M \subseteq \{1,\ \ldots,\ \mathsf{arity}\ p\}\}$$

until a fixpoint is reached.

Example For *append*, the allowed modes are inferred as follows:

$\{\}$ is illegal, because it is impossible to compute the value of *ys* in the first clause

$\{1\}$ is illegal for the same reason

$\{2\}$ is illegal, because it is impossible to compute the value of x in the second clause

$\{3\}$ is legal, because

- in the first clause, we can compute the first and second argument $(Nil,\ ys)$ from the third argument *ys*
- in the second clause, we can compute x and *zs* from the third argument. By recursively calling *append* with mode $\{3\}$, we can compute the value of *xs* and *ys*. Thus, we also know the value of the first and second argument $(Cons\ x\ xs,\ ys)$.

$\{1,\ 2\}$ is legal, because

- in the first clause, we can compute the third argument *ys* from the first and second argument
- in the second clause, we can compute x, *xs* and *ys* from the first and second argument. By recursively calling *append* with mode $\{1,\ 2\}$, we can compute the value of *zs*. Thus, we also have the value of the third argument *Cons x zs*

$\{1,\ 3\}$, $\{2,\ 3\}$, $\{1,\ 2,\ 3\}$ are legal as well (see e.g. $\{3\}$)

Well-moded queries A query $(t_1, \ldots, t_n) \in p$ is called *well-moded* with respect to a mode assignment *modes* iff

$$\{i \mid t_i \text{ is not a variable}\} \in modes\ p$$

Mixing predicates and functions The above conditions for the consistency of modes are sufficient, if the only functions occurring in the clauses are *constructor functions.* If we allow arbitrary functions to occur in the clauses, we have to impose some additional restrictions on the positions of their occurrence. Since non-constructor functions may not be inverted, they cannot appear in an *input position* in the clause head or in an *output position* in the clause body. Thus, we rephrase conditions (1) and (2) to

$(1')$ $v_0 = \mathsf{vars\_of}\ (\mathsf{args\_of}\ \{i \in M \mid \mathsf{funs\_of}\ t_i \subseteq \mathcal{C}\}\ (t_1,\ \ldots,\ t_k)) \wedge$
$\forall i \in M.\ \mathsf{funs\_of}\ t_i \not\subseteq \mathcal{C} \longrightarrow \mathsf{eqtype}\ t_i$

$(2')$ $\forall 1 \leq i \leq m.\ \exists M' \in modes\ q_{\pi(i)}.$
$M' \subseteq \mathsf{known\_args}\ v_{i-1}\ (u_1^{\pi(i)},\ \ldots,\ u_{n_{\pi(i)}}^{\pi(i)}) \wedge$
$\mathsf{funs\_of}\ (\mathsf{args\_of}\ (\{1, \ldots, \mathsf{arity}\ q_{\pi(i)}\} \backslash M')\ (u_1^{\pi(i)},\ \ldots,\ u_{n_{\pi(i)}}^{\pi(i)})) \subseteq \mathcal{C}$

where $\mathcal{C}$ is the set of constructor functions and $\mathsf{funs\_of}$ returns the set of all functions occurring in a tuple. The intuition behind ($1'$) is as follows: if some of the input parameters specified by M contain non-constructor functions, we try mode analysis with a subset of M that does not contain the problematic input parameters. After successful execution, we compare the computed values of t_j, where $j \in M \wedge \mathsf{funs\_of}\ t_j \not\subseteq \mathcal{C}$, with the values provided as input arguments to the predicate. For this to work properly, the terms t_j need to have an *equality type*, i.e. not be of a function type or a datatype involving function types. Note that any M_2 with $M_1 \subseteq M_2$ will be a valid mode, provided M_1 is a valid mode and

$$\forall j \in M_2 \backslash M_1.\ \mathsf{funs\_of}\ t_j \not\subseteq \mathcal{C} \longrightarrow \mathsf{eqtype}\ t_j$$

As condition ($2'$) suggests, we can get around the restriction on the occurrence of non-constructor functions in the clause body by choosing modes M' which are sufficiently large, i.e. have sufficiently many input parameters.

6.3.2 Translation scheme

In the following section, we will explain how to translate predicates given by a set of Horn Clauses into functional programs in the language ML. For each legal mode of a predicate, a separate function will be generated. Given a tuple of input arguments, a predicate may return a potentially infinite sequence of result tuples. Sequences are represented by the type `'a seq` which supports the following operations:

```
Seq.empty  : 'a seq
Seq.single : 'a -> 'a seq
Seq.append : 'a seq * 'a seq -> 'a seq
Seq.map    : ('a -> 'b) -> 'a seq -> 'b seq
Seq.flat   : 'a seq seq -> 'a seq
Seq.pull   : 'a seq -> ('a * 'a seq) option
```

Note that sequences are implemented *lazily*, i.e. the evaluation of elements is delayed. The evaluation of the head element can be forced using the function `Seq.pull`, which either returns the head element and the tail of the sequence, or the element `None` if the sequence is empty. More information on how to implement such lazy data structures in an eager language such as ML can be found e.g. in the book by Paulson [91]. In the sequel, we will write `s1 ++ s2` instead of `Seq.append (s1, s2)`. In addition, we define the operator

```
fun s :-> f = Seq.flat (Seq.map f s);
```

which will be used to compose subsequent calls of predicates. In order to embed additional side conditions, i.e. boolean expressions, into this sequence programming scheme, we define the function

```
fun ?? b = if b then Seq.single () else Seq.empty;
```

returning either a singleton sequence containing just the unit element `()` if `b` evaluates to `true`, or the empty sequence if `b` evaluates to `false`. Conversely, if we want to use a sequence in an ordinary boolean expression, it is useful to have a function

```
fun ?! s = (case Seq.pull s of None => false | Some _ => true);
```

that checks for the non-emptiness of the sequence `s`. Using the above operators on sequences, the modes {1, 2} and {3} of predicate *append* can be translated into the ML functions

```
append_1_2 : 'a list * 'a list -> 'a list seq
append_3   : 'a list -> ('a list * 'a list) seq
```

which are defined as follows:

```
fun append_1_2 inp =
  Seq.single inp :->
    (fn (Nil, ys) => Seq.single (ys) | _ => Seq.empty) ++
  Seq.single inp :->
    (fn (Cons (x, xs), ys) =>
      append_1_2 (xs, ys) :->
        (fn (zs) => Seq.single (Cons (x, zs)) | _ => Seq.empty)
      | _ => Seq.empty);

fun append_3 inp =
  Seq.single inp :->
    (fn (ys) => Seq.single (Nil, ys) | _ => Seq.empty) ++
  Seq.single inp :->
    (fn (Cons (x, zs)) =>
      append_3 (zs) :->
        (fn (xs, ys) => Seq.single (Cons (x, xs), ys)
          | _ => Seq.empty)
      | _ => Seq.empty);
```

In the above translation, every operand of `++` corresponds to one clause of the predicate. Initially, the input is converted into a one-element sequence using `Seq.single`, to which successively all predicates in the body of the clause are applied using `:->`. Therefore, the operator `:->` can also be interpreted as a visualization of dataflow.

We will now describe the general translation scheme. Assume the predicate to be translated has the clause

$$(ipat_1,\ opat_1) \in q_1 \Longrightarrow \ldots \Longrightarrow (ipat_m,\ opat_m) \in q_m \Longrightarrow (ipat_0,\ opat_0) \in p$$

To simplify notation, we assume without loss of generality that the predicates in the body of p are already sorted with respect to the permutation π calculated during mode analysis and that the arguments of the predicates are already partitioned into input arguments $ipat_i$ and output arguments $opat_i$. Then, p is translated into the function

```
fun $p$ inp =
  Seq.single inp :->
    (fn $ipat_0$ => $q_1$ $ipat_1$ :->
        (fn $opat_1$ => $q_2$ $ipat_2$ :->
            $\ddots$
                (fn $opat_m$ => Seq.single $opat_0$
                  | _ => Seq.empty)
              $\vdots$
          | _ => Seq.empty)
      | _ => Seq.empty)
  ++
  ...;
```

where the ... after the operator `++` correspond to the translation of the remaining clauses of p. A characteristic feature of this translation is the usage of ML's built-in pattern matching mechanism instead of unification and logical variables. Before calling a predicate q_i in the body of the clause, the output pattern $opat_{i-1}$ of the preceeding predicate is checked. Before calling the first predicate q_1, the input pattern $ipat_0$ in the head of the clause is checked.

Relation to Haskell list comprehensions The above translation scheme is reminiscent of the translation scheme for *list comprehensions*, which is described in the report on the Haskell programming language [57, §3.11] and implemented in most Haskell compilers. Using list comprehensions, the functions append_1_2 and append_3 could be written in Haskell in a more compact way as follows:

```
append_1_2 inp =
  [ys | ([], ys) <- [inp]] ++
  [x : zs | (x : xs, ys) <- [inp], zs <- append_1_2 (xs, ys)]

append_3 inp =
  [([], ys) | ys <- [inp]] ++
  [(x : xs, ys) | x : zs <- [inp], (xs, ys) <- append_3 zs]
```

The general way of formulating a predicate using list comprehension notation is

```
p inp =
  [opat_0 | ipat_0 <- [inp], opat_1 <- q_1 ipat_1, ..., opat_m <- ipat_m]
  ++
  ...
```

Example: executing β-reduction As an example of a program making use of both functional and logical features, we now consider the specification of β-reduction for λ-terms in de Bruijn notation, which was introduced in §5.4.1. The specification of the lifting and substitution functions *lift* and *subst* used in the definition of $\rightarrow_\beta$ is purely functional. Their translation to ML is straightforward and is therefore not shown here. The specification of β-reduction is essentially a functional logic program. Using the translation scheme described above, the HOL specification of $\rightarrow_\beta$ can be translated to the following ML program for mode $\{1\}$, which, given a term s, computes the sequence of all terms t with $s \rightarrow_\beta t$:

```
fun beta_1 inp =
  Seq.single inp :->
    (fn (App (Abs s, t)) =>
      Seq.single (subst s t 0) | _ => Seq.empty) ++
  Seq.single inp :->
    (fn (App (s, u)) =>
      beta_1 (s) :->
        (fn (t) => Seq.single (App (t, u)) | _ => Seq.empty)
      | _ => Seq.empty) ++
  Seq.single inp :->
    (fn (App (u, s)) =>
      beta_1 (s) :->
        (fn (t) => Seq.single (App (u, t)) | _ => Seq.empty)
      | _ => Seq.empty) ++
  Seq.single inp :->
    (fn (Abs s) =>
      beta_1 (s) :->
        (fn (t) => Seq.single (Abs t) | _ => Seq.empty)
      | _ => Seq.empty);
```

Note that the recursive function `subst` can easily be called from within the logic program `beta_1`.

Running the translated program We will now try out the compiled predicate on a small example: the sequence

```
val test = beta_1 (Abs (Abs (App
  (Abs (App (App (Var 2, Var 0), Var 0)),
   App (Abs (App (App (Var 2, Var 0), Var 0)), Var 0)))));
```

contains the possible reducts of the term $\lambda f\ x.\ (\lambda y.\ f\ y\ y)\ ((\lambda z.\ f\ z\ z)\ x)$. The first element of this sequence is

```
> Seq.hd test;
val it = Abs (Abs (App
  (App (Var 1, App (Abs (App (App (Var 2, Var 0), Var 0)), Var 0)),
   App (Abs (App (App (Var 2, Var 0), Var 0)), Var 0))))
```

which denotes the term $\lambda f\ x.\ f\ ((\lambda z.\ f\ z\ z)\ x)\ ((\lambda z.\ f\ z\ z)\ x)$. There is yet another solution for our query, namely

```
> Seq.hd (Seq.tl test);
val it = Abs (Abs (App
  (Abs (App (App (Var 2, Var 0), Var 0)),
   App (App (Var 1, Var 0), Var 0))))
```

which corresponds to the term $\lambda f\ x.\ (\lambda y.\ f\ y\ y)\ (f\ x\ x)$.

6.3.3 Extending the mode system

The mode system introduced in §6.3.1 is not always sufficient: For example, it does not cover inductive relations such as the transitive closure

inductive
$(x,\ x) \in \mathit{rtrancl}\ r$
$(x,\ y) \in r \Longrightarrow (y,\ z) \in \mathit{rtrancl}\ r \Longrightarrow (x,\ z) \in \mathit{rtrancl}\ r$

which take other inductive relations as arguments. This case can be covered by introducing so-called *higher-order modes*: a mode of a higher-order relation p taking n relations $r_1, \ldots, r_l$ as arguments and returning a relation as result is an $n+1$ tuple, where the first n components of the tuple correspond to the modes of the argument relations, and the last component corresponds to the mode of the resulting relation, i.e.

$$\mathit{modes}\ p \subseteq \mathcal{P}(\{1,\ldots,\mathsf{arity}\ r_1\}) \times \cdots \times \mathcal{P}(\{1,\ldots,\mathsf{arity}\ r_l\}) \times \mathcal{P}(\{1,\ldots,\mathsf{arity}\ p\})$$

For example, *rtrancl* has modes $\{(\{1\},\{1\}),(\{2\},\{2\}),(\{1,2\},\{1,2\})\}$, i.e. if r has mode $\{1\}$ then *rtrancl* r has mode $\{1\}$ as well. A higher-order relation may have clauses of the form

$$(u_1^1,\ldots,u_{n_1}^1) \in Q_1 \Longrightarrow \ldots \Longrightarrow (u_1^m,\ldots,u_{n_m}^m) \in Q_m \Longrightarrow (t_1,\ldots,t_k) \in p\ r_1 \ldots r_l$$

where $Q_{i'} = r_i \mid q_j\ Q'_{\varrho_1}\ \ldots\ Q'_{\varrho_{l'}}$

To describe the consistency of a higher order mode $(M_1,\ \ldots,\ M_l,\ M)$ with respect to a mode assignment *modes* and the above clause, we rephrase condition (2) of the definition of consistency given in §6.3.1 to

$$(2')\quad \forall 1 \le i \le m.\ \exists M' \in \mathit{modes}'\ Q_{\pi(i)}.\ M' \subseteq \mathsf{known\_args}\ v_{i-1}\ (u_1^{\pi(i)},\ \ldots,\ u_{n_{\pi(i)}}^{\pi(i)})$$

where
$\mathit{modes}'\ r_i = \{M_i\}$
$\mathit{modes}'\ (q_j\ Q'_{\varrho_1}\ \ldots\ Q'_{\varrho_{l'}}) = \{M' \mid \exists M'_1 \in \mathit{modes}'\ Q'_{\varrho_1} \ldots M'_{l'} \in \mathit{modes}'\ Q'_{\varrho_{l'}}.$
$(M'_1,\ \ldots,\ M'_{l'},\ M') \in \mathit{modes}\ q_j\}$

Mode $(\{1\},\{1\})$ of *rtrancl* could be translated as follows

```
fun rtrancl_1__1 r inp =
  Seq.single inp ++ r inp :-> rtrancl_1__1 r;
```

Analogously, the translation of mode $(\{2\},\{2\})$ is

```
fun rtrancl_2__2 r inp =
  Seq.single inp ++ rtrancl_2__2 r inp :-> r;
```

We can then use `rtrancl_1__1` to define a function for computing the set of solutions to the query $s \longrightarrow_\beta^* X \wedge \neg(\exists u.\ X \longrightarrow_\beta u)$ as follows:

```
fun nf inp =
  Seq.single inp :->
    (fn s => rtrancl_1__1 beta_1 s :->
      (fn t => ?? (not (?! (beta_1 t))) :->
        (fn () => Seq.single t)));
```

In other words, `nf` searches for a normal form of the term s. For example, the normal form $\lambda f\ x.\ f\ (f\ x\ x)\ (f\ x\ x)$ of the term $\lambda f\ x.\ (\lambda y.\ f\ y\ y)\ ((\lambda z.\ f\ z\ z)\ x)$ from the above example can be computed by

```
> Seq.hd (nf (Abs (Abs (App
  (Abs (App (App (Var 2, Var 0), Var 0)),
   App (Abs (App (App (Var 2, Var 0), Var 0)), Var 0))))));
val it = Abs (Abs (App (App (Var 1, App
  (App (Var 1, Var 0), Var 0)), App (App (Var 1, Var 0), Var 0))))
```

Note that the standard definition of the transitive closure from the Isabelle/HOL library, which was presented in §4.3.5, cannot be used for this purpose. When executing the rule *rtrancl-into-rtrancl* for the step case with the mode $(\{1\},\{1\})$, the premise $(a,\ b) \in r^*$ containing the recursive call is evaluated before the premise $(b,\ c) \in r$, which leads to nontermination.

6.3.4 Discussion

We briefly discuss some problems encountered when translating specifications into a functional programming language, which concern the *completeness* of the translation scheme described in the previous sections.

Termination of logic programs A source of possible nontermination is the Prolog-style *depth-first* execution strategy of the translated inductive relations. Breadth-first search is much trickier to implement, but a *depth-first iterative deepening* strategy could easily be implemented by giving the generated functions an additional integer argument denoting the recursion depth, and cutting off the search if this depth exceeds a given limit. Moreover, some inferred modes or permutations of predicates in the body of a clause may turn out to be non-terminating (see e.g. the remark in §6.3.3). Termination of logic programs is an interesting problem in its own right, which we do not attempt to solve here. A good survey of this topic is given by de Schreye and Decorte [32]. Several algorithms for analyzing the termination of logic programs have been proposed by Lindenstrauss and Sagiv [62], as well as Plümer [101], to name just a few examples.

Mode system The somewhat coarse distinction between *input* and *output* arguments made by the mode system given in §6.3.1 rules out specifications where actual unification of partially instantiated data structures instead of just matching is required to synthesize results. For example, it is impossible to find a suitable mode for the typing judgement $e \vdash t : T$ given in §5.4.2. The mode $\{1, 2\}$, which means that T has to be computed from e and t, is illegal since we would have to "guess" the argument type T when executing $e \vdash Abs\ t : (T \Rightarrow U)$. Even the mode $\{1, 2, 3\}$, which means that also the expected type of the term is already given, does not work since when executing $e \vdash (s \circ t) : U$, we cannot deduce the type $T \Rightarrow U$ of the term s and hence cannot execute any of the premises of the typing rule for the application case.

Eager evaluation Another problem is due to ML's *eager evaluation* strategy. For example,

defs
$g \equiv \lambda x\ y.\ x$
$f_1 \equiv \lambda x.\ g\ x\ (hd\ [])$
recdef
$f_2\ 0 = 0$
$f_2\ (Suc\ x) = g\ (f_2\ x)\ (f_2\ (Suc\ x))$

is an admissible HOL specification, but, if compiled naively, f_1 raises an exception, because the argument [] cannot be handled by hd, and f_2 loops. To avoid this, the definition of g could be expanded, or the critical arguments could be wrapped into dummy functions, to delay evaluation. Paulin-Mohring and Werner [88] discuss this problem in detail and also propose alternative target languages with *lazy evaluation*.

6.4 Related work

Previous work on executing HOL specifications There has already been some work on generating executable programs from specifications written in HOL. One of the first papers on this topic is by Rajan [104] who translates HOL datatypes and recursive functions to ML. However, inductive definitions are not covered in this paper. Andrews [5] has chosen λProlog as a target language. His translator for a higher order specification language called S can also handle specifications of transition rules of programming languages such as CCS, although these are given in a somewhat different way than the inductive definitions of Isabelle/HOL. In contrast to our approach, all functions have to be translated into predicates in order to be executable by λProlog. On the other hand it is possible to execute a wider range of specifications and queries, as λProlog allows embedded universal quantifiers and implications and supports higher-order unification. Although quite efficient implementations of λProlog have been devised recently, the underlying execution strategy is comparable in complexity to a fully-fledged theorem prover. Similar comments apply to the logical framework Elf [94], which also provides a higher-order logic programming language.

Other theorem provers Aagaard et al [1] introduce a functional language called fl, together with a suitable theorem prover. Thanks to a "lifting" mechanism, their system supports both *execution* of fl functions as well as *reasoning* about fl functions in a seamless way.

In §5.1.2, we have already discussed the *Coq* [12] theorem prover based on the *Calculus of Inductive Constructions*, and its philosophy of distinguishing between computable and non-computable objects. This is in contrast to HOL, where specifications often tend to mix such objects in an arbitrary manner. Coq can directly generate Haskell and OCaml code from definitions of recursive datatypes and functions such as *nat* and *add* from §6.2. However, in contrast to our framework, there is no possibility to *directly* interpret specifications such as *append* as a logic program.

The latest version of the theorem prover *PVS* [84] includes a procedure for evaluating ground terms. The PVS ground evaluator essentially consists of a translator from an executable subset of PVS into Common Lisp. The unexecutable fragments are uninterpreted functions, non-bounded quantification and higher-order equalities.

The *Centaur* system [55] is an environment for specifying programming languages. One of its components is a Prolog-style language called *Typol* [33], in which transition rules of natural semantics can be specified. Attali et al [6] have used Typol to specify a formal, executable semantics of a large subset of the programming language *Java*. Originally, Typol specifications were compiled to Prolog in order to execute them. Recently, Dubois and Gayraud [35] have proposed a translation of Typol specifications to ML. The consistency conditions for modes described in §6.3.1 are inspired by this paper.

Languages for combining functional and logic programming There are numerous different approaches for the combination of functional and logic programming and we mention only a few typical ones. The programming language Curry [44] uses *narrowing* as an execution model. The programming language Mercury [112] has a very rich mode system and uses mode analysis to generate efficient C code.

Chapter 7

Conclusion

7.1 Achievements

In this thesis, we have presented an extension of the higher order logical framework Isabelle with primitive proof terms. The developed infrastructure includes algorithms for synthesizing proof terms via higher order resolution, compressing proofs by omitting redundant syntactic information, and reconstructing omitted information in compressed proofs. Moreover, we have also looked at the problem of proof generation for equational logic, and proposed a new and improved strategy for contextual rewriting.

Our work increases the reliability of Isabelle, and contributes to a better understanding of the theoretical foundations of its kernel. It also forms an important basis for future applications such as proof-carrying code.

The proof term calculus introduced in this thesis has been used to build a generic framework for the extraction of programs from constructive proofs. This system, which, to our knowledge, is the first one of this kind for a theorem prover of the HOL family, has been applied successfully to several nontrivial case studies.

Finally, we have investigated an alternative approach for obtaining programs from specifications by directly interpreting inductively defined predicates as logic programs. For this purpose, we have developed a lightweight mechanism for translating logic programs into functional programs.

With the introduction of program extraction and Prolog-style execution of inductive predicates, we have come closer to the vision of Isabelle as an integrated environment for *specification*, *proving* and *programming*. Moreover, the program extraction framework is very likely to make Isabelle more attractive for users from communities interested in constructive logic.

7.2 Future work

To conclude, we describe some ideas for extending the work presented in this thesis.

Development of constructive proofs Now that a stable infrastructure for program extraction is available in Isabelle, which has been successfully applied to first examples, the next step is to tackle some more advanced case studies. Good candidates seem to be algorithms from graph theory. For example, one could think of extracting a graph colouring algorithm from the proof of the *Five Colour Theorem* by Bauer and Nipkow [14]. Unfortunately, although the main proof is essentially constructive, since it is by induction on the size of the graph, some other parts of the proof, which contribute to the computational content, are not. For example, the construction of the graph colouring relies on the fact that each *near triangulation* contains a vertex with degree ≤ 5, which is proved by contradiction [14, §3.2].

Another promising area for the application of program extraction is that of *constructive analysis*, which was pioneered by Bishop and Bridges [22]. To date, all of the formalizations of analysis that have been done in the HOL system and Isabelle are classical, and many standard results of classical analysis, such as the *intermediate value theorem*, need to be reformulated in order to be provable constructively. It is an interesting research project to examine how difficult such a constructive reformulation would be compared to the original classical formulation, and what its benefits are.

Program extraction from classical proofs Recently, there has been considerable interest in extending program extraction to classical proofs. Most of the approaches to this problem are based on transforming classical proofs into constructive ones, usually by applying a variant of the *double negation translation*. As has been mentioned in §5.3, a naive translation can lead to huge programs and is therefore infeasible. This observation has given rise to several refined methods for program extraction from classical proofs, such as the one proposed by Berger, Buchholz and Schwichtenberg [17]. Often, there are subtle differences between these methods with respect to their efficiency and the class of formulae covered by them. A future research project could consist in assessing the feasibility of applying such methods to realistic classical proofs in Isabelle/HOL, and in finding convincing applications for them.

Program extraction for other logics Another important point to study is how our framework for program extraction can be instantiated to other logics, such as constructive versions of Zermelo-Fränkel Set Theory (ZF). For the HOL instantiation described in §4.3, matters were particularly simple, since HOL and Isabelle's meta logic share the same type system. This is in contrast to ZF, which is essentially untyped and simulates the concept of type checking by explicit logical reasoning about set membership statements.

Optimizing proof procedures Although we have described several quite effective methods for reducing the size of proofs by eliminating syntactic redundancies, these methods are not a panacea. For example, practical experience has shown that many of the proof procedures for arithmetic in Isabelle/HOL tend to produce huge proofs even for relatively trivial formulae. In the quotient and remainder example discussed in §5.1, more than 85% of the proof term produced by Isabelle consisted of lengthy computations on numerals, although the arithmetic expressions involved were not particularly complicated. Such kinds of inefficiencies in proofs usually cannot be addressed simply by applying proof compression algorithms as described in §2.4.2, or proof rewrite rules for eliminating detours such as those given in §2.2.2. Instead, a more fundamental redesign of proof procedures is required, taking the size of the produced proofs into account. The problem of inefficient proof procedures is also well-known to implementors of proof assistants based on type theory, such as Coq. Sacerdoti Coen [105] calls this the problem of "overkilling tactics" and gives several suggestions for reducing the size of proofs generated by Coq's procedure for associative-commutative rewriting on rings.

Reflection By generating executable code from specifications, computations that would otherwise have to be done inside the theorem prover, e.g. via term rewriting, can be performed more efficiently in the target language. An important question is how results computed by an external execution of generated code can be reimported into the theorem prover in a safe way, which is sometimes referred to as *reflection*. Harrison [47] gives a good overview of different kinds of reflection and discusses their integration into an LCF-style theorem prover. However, his conclusion is that both the theoretical foundations and the practical applicability of reflection mechanisms still need to be studied in more detail. It also seems interesting to examine how reflection relates to the approach of *normalization by evaluation* put forward by Berger and Schwichtenberg [18], where the evaluation mechanism of the implementation language of a theorem prover is used to efficiently perform operations like β-reduction of λ-terms.

Sharing theories between theorem provers The infrastructure for proof terms developed in this thesis can also be used as a basis for exploring the possibilities of sharing theories, i.e. collections of axioms, definitions and theorems together with their proofs, between different theorem proving systems. Importing theories developed in Isabelle/HOL into the HOL system (or vice versa) is relatively easy at least from the point of view of the underlying logics, although the technical challenges involved in the development of such a translation mechanism should not be underestimated. In contrast, sharing theories between Isabelle/HOL and systems based on constructive type theories is much more complicated, since the classical and constructive worlds have to be reconciled in some way. As already mentioned earlier, the careless addition of certain classical axioms to constructive type theories can give rise to quite subtle inconsistencies [40]. As a first step towards linking theorem provers based on classical and constructive logics, Felty and Howe have proposed a method for hybrid interactive theorem proving using Nuprl and HOL [36]. A central part of their approach is a new set-theoretic semantics for Nuprl, which allows an embedding of HOL's classical type theory. It is important to note that Felty and Howe do not consider the translation of *proofs* from HOL to Nuprl, but simply rely on the correctness of imported theorems. As a future research project, it would therefore be interesting to study how this method can be extended to cover the translation of proofs, and to examine similar embeddings of HOL in other theorem provers based on type theory, such as Coq. To avoid the addition of classical axioms, one might also consider the restriction of such a translation mechanism to a constructive fragment of HOL similar to the one used in §4.3 for the purpose of program extraction.

List of Figures

Bibliography

[1] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted-FL: A Pragmatic Implementation of Combined Model Checking and Theorem Proving. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference (TPHOLs'99)*, volume 1690 of *Lecture Notes in Computer Science*, pages 323–340. Springer-Verlag, 1999.

[2] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization.* PhD thesis, University of Edinburgh, November 1993.

[3] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In Jan F. Groote Marc Bezem, editor, *Typed Lambda Calculi and Applications, International Conference (TLCA '93)*, volume 664 of *LNCS*, pages 13–28. Springer-Verlag, 1993.

[4] Penny Anderson. Program extraction in a logical framework setting. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, volume 822 of *LNAI*, pages 144–158. Springer-Verlag, July 1994.

[5] James H. Andrews. Executing formal specifications by translation to higher order logic programming. In Elsa L. Gunter and Amy Felty, editors, *10th International Conference on Theorem Proving in Higher Order Logics*, volume 1275 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, 1997.

[6] Isabelle Attali, Denis Caromel, and Marjorie Russo. A formal and executable semantics for Java. In *Proceedings of Formal Underpinnings of Java, an OOPSLA'98 Workshop, Vancouver, Canada*, 1998. Technical report, Princeton University.

[7] Abdelwaheb Ayari and David Basin. A higher-order interpretation of deductive tableau. *Journal of Symbolic Computation*, 31(5):487–520, May 2001.

[8] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[9] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[10] Henk Barendregt and Herman Geuvers. Proof assistants using dependent type systems. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 18, pages 1149 – 1238. Elsevier Science Publishers, 2001.

[11] Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives.* Thèse de doctorat, Université Paris 7, November 1999.

[12] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq proof assistant reference manual – version 7.2. Technical Report 0255, INRIA, February 2002.

[13] Bruno Barras and Benjamin Werner. Coq in Coq. To appear in Journal of Automated Reasoning.

[14] Gertrud Bauer and Tobias Nipkow. The 5 colour theorem in Isabelle/Isar. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, volume 2410 of *LNCS*, pages 67–82. Springer-Verlag, 2002.

[15] Holger Benl, Ulrich Berger, Helmut Schwichtenberg, Monika Seisenberger, and Wolfgang Zuber. Proof theory at work: Program development in the Minlog system. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume II: Systems and Implementation Techniques of *Applied Logic Series*, pages 41–71. Kluwer Academic Publishers, Dordrecht, 1998.

[16] Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan F. Groote, editors, *Typed Lambda Calculi and Applications, International Conference (TLCA '93)*, pages 91–106, Berlin, Germany, 1993. Springer-Verlag.

[17] Ulrich Berger, Wilfried Buchholz, and Helmut Schwichtenberg. Refined program extraction from classical proofs. *Annals of Pure and Applied Logic*, 114:3–25, 2002.

[18] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In R. Vemuri, editor, *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society Press, 1991.

[19] Ulrich Berger, Helmut Schwichtenberg, and Monika Seisenberger. The Warshall algorithm and Dickson's lemma: Two examples of realistic program extraction. *Journal of Automated Reasoning*, 26:205–221, 2001.

[20] Stefan Berghofer and Tobias 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 *LNCS*. Springer-Verlag, 2002.

[21] Stefan Berghofer and Markus Wenzel. Inductive datatypes in HOL — lessons learned in Formal-Logic Engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *LNCS*. Springer-Verlag, 1999.

[22] Errett Bishop and Douglas S. Bridges. *Constructive Analysis*. Springer-Verlag, 1985.

[23] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 329–344. Elsevier, 1998. http://www.elsevier.nl/locate/entcs/volume15.html.

[24] Richard J. Boulton. Transparent optimisation of rewriting combinators. *Journal of Functional Programming*, 9(2):113–146, March 1999.

[25] Luitzen Egbertus Jan Brouwer. Intuïtionistische splitsing van mathematische grondbegrippen. *Nederl. Akad. Wetensch. Verslagen*, 32:877–880, 1923.

[26] Martin David Coen. *Interactive program derivation.* PhD thesis, Cambridge University, November 1992.

[27] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, NJ, 1986.

[28] Thierry Coquand. *Une Théorie des Constructions.* PhD thesis, Université Paris 7, January 1985.

[29] Thierry Coquand and Daniel Fridlender. A proof of Higman's lemma by structural induction. Unpublished draft, available at `http://www.math.chalmers.se/~frito/Papers/open.ps.gz`, November 1993.

[30] Luís Cruz-Filipe and Bas Spitters. Program extraction from large proof developments. In David Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics, 16th International Conference*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[31] N. G. de Bruijn. A survey of the project AUTOMATH. In J. R. Hindley and J. P. Seldin, editors, *Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 580–606. Academic Press, London, 1980.

[32] Danny De Schreye and S Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.

[33] Thierry Despeyroux. Typol: a formalism to implement natural semantics. Technical Report 94, INRIA, 1988.

[34] Gilles Dowek. A Complete Proof Synthesis Method for the Cube of Type Systems. In Gérard Huet, Gordon Plotkin, and Claire Jones, editors, *Proceedings of the second workshop on Logical Frameworks, Edinburgh*, pages 135–163, 1991.

[35] Catherine Dubois and Richard Gayraud. Compilation de la sémantique naturelle vers ML. In *Proceedings of journées francophones des langages applicatifs (JFLA99)*, 1999. Available via `http://pauillac.inria.fr/~weis/jfla99/ps/dubois.ps`.

[36] Amy P. Felty and Douglas J. Howe. Hybrid interactive theorem proving using Nuprl and HOL. In William McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, volume 1249 of *Lecture Notes in Computer Science*, pages 351–365. Springer-Verlag, 1997.

[37] Maribel Fernández and Paula Severi. An operational approach to program extraction in the Calculus of Constructions. In *International Workshop on Logic Based Program Development and Transformation (LOPSTR'02)*, LNCS. Springer, 2002.

[38] Daniel Fridlender. Higman's lemma in type theory. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs, International Workshop TYPES'96*, volume 1512 of *Lecture Notes in Computer Science*, pages 112–133. Springer-Verlag, 1998.

[39] Gérard Huet. The undecidability of unification in third order logic. *Information and Control*, 22(3):257–367, 1973.

[40] Herman Geuvers. Inconsistency of classical logic in type theory. Unpublished note, available at http://www.cs.kun.nl/~herman/note.ps.gz, November 2001.

[41] Herman Geuvers, Randy Pollack, Freek Wiedijk, and Jan Zwanenburg. The algebraic hierarchy of the FTA Project. *Journal of Symbolic Computation, Special issue on the Integration of Automated Reasoning and Computer Algebra Systems*, pages 271–286, 2002.

[42] Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[43] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[44] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[45] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[46] John Harrison. HOL Done Right. Unpublished draft, available at http://www.cl.cam.ac.uk/users/jrh/papers/holright.html, August 1995.

[47] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. Available on the Web as http://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz.

[48] S. Hayashi and H. Nakano. *PX, a Computational Logic*. Foundations of Computing. MIT Press, 1988.

[49] Arend Heyting. *Mathematische Grundlagenforschung. Intuitionismus. Beweistheorie.* Springer, 1934.

[50] G. Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(2):326–336, 1952.

[51] William A. Howard. The formulae-as-types notion of construction. In J. R. Hindley and J.P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[52] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, June 1975.

[53] Isabelle/HOL theory library. http://isabelle.in.tum.de/library/HOL/.

[54] Paul Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1995. Technical Report TR 95-1509.

[55] Ian Jacobs and Laurence Rideau-Gallot. A Centaur tutorial. Technical Report 140, INRIA Sophia-Antipolis, July 1992.

[56] Felix Joachimski and Ralph Matthes. Short proofs of normalization for the simply-typed λ-calculus, permutative conversions and Gödel's T. *Archive for Mathematical Logic*, 42(1):59–87, 2003.

[57] Simon Peyton Jones and John Hughes. Report on the programming language Haskell 98, February 1999.

[58] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, June 2000.

[59] S.C. Kleene. *Introduction to Metamathematics.* North Holland, 1952.

[60] Andrei Kolmogorov. Zur Deutung der intuitionistischen Logik. *Mathematische Zeitschrift*, 35:58–65, 1932.

[61] Pierre Letouzey. A new Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *LNCS*, pages 200–219. Springer-Verlag, 2003.

[62] Naomi Lindenstrauss and Yehoshua Sagiv. Automatic Termination Analysis of Logic Programs. In Lee Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 63–77. MIT Press, 1997.

[63] Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[64] Marko Luther. More on Implicit Syntax. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *Proceedings of the First International Joint Conference on Automated Reasoning (IJCAR 2001), Siena*, volume 2083 of *LNAI*, pages 386–400. Springer-Verlag, 2001.

[65] Lena Magnusson. *The Implementation of ALF—a Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* Phd thesis, Dept. of Computing Science, Chalmers Univ. of Technology and Univ. of Göteborg, 1994.

[66] William McCune. OTTER 2.0. In Mark Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, Vol. 449*, pages 663–664, New York, July 1990. Springer-Verlag. Extended abstract.

[67] James McKinna and Rod M. Burstall. Deliverables: A categorical approach to program development in type theory. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium*, volume 711 of *LNCS*, pages 32–67, Gdansk, Poland, 30 August– 3 September 1993. Springer.

[68] C. S. Mellish. The automatic generation of mode declarations for Prolog programs. Technical Report 163, Department of Artificial Intelligence, University of Edinburgh, August 1981.

[69] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[70] Alexandre Miquel. The Implicit Calculus of Constructions. In Samson Abramsky, editor, *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications (TLCA 2001)*, volume 2044 of *Lecture Notes in Computer Science*, pages 344–359. Springer-Verlag, 2001.

[71] Chetan Murthy. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell University, 1990.

[72] Chetan R. Murthy and James R. Russell. A constructive proof of Higman's lemma. In John C. Mitchell, editor, *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*, pages 257–269, Philadelphia, PA, June 1990. IEEE Computer Society Press.

[73] C. Nash-Williams. On well-quasi-ordering finite trees. *Proceedings of the Cambridge Philosophical Society*, 59(4):833–835, 1963.

[74] George Necula. A scalable architecture for Proof-Carrying Code. In Herbert Kuchen and Kazunori Ueda, editors, *Functional and Logic Programming, 5th International Symposium, FLOPS 2001, Tokyo, Japan, March 7-9, 2001, Proceedings*, volume 2024 of *Lecture Notes in Computer Science*. Springer, 2001.

[75] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, New York, 1997.

[76] George C. Necula. *Compiling with Proofs.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1998.

[77] George C. Necula and Peter Lee. Efficient representation and validation of proofs. In *13th IEEE Symp. Logic in Computer Science (LICS'98)*, pages 93–104. IEEE Computer Society Press, 1998.

[78] Tobias Nipkow. Proof transformations for equational theories. In *Proc. 5th IEEE Symp. Logic in Computer Science*, pages 278–288, 1990.

[79] Tobias Nipkow. Functional unification of higher-order patterns. In *8th IEEE Symp. Logic in Computer Science*, pages 64–74. IEEE Computer Society Press, 1993.

[80] Tobias Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.

[81] Tobias Nipkow. More Church-Rosser proofs (in Isabelle/HOL). *Journal of Automated Reasoning*, 26:51–66, 2001.

[82] Tobias Nipkow. Structured Proofs in Isar/HOL. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, volume 2646 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 2003.

[83] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[84] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS System Guide version 2.3. Technical report, SRI International Computer Science Laboratory, Menlo Park CA, September 1999.

[85] Christine Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM.

[86] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* Thèse d'université, Paris 7, January 1989.

[87] Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.

[88] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[89] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[90] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[91] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[92] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

[93] Lawrence C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Journal of Logic and Computation*, 7(2):175–204, April 1997.

[94] Frank Pfenning. Logic programming in the LF Logical Framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 66–78. Cambridge University Press, 1991.

[95] Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *6th IEEE Symposium on Logic in Computer Science*, pages 74–85. IEEE Computer Society Press, 1991.

[96] Frank Pfenning. Logical and meta-logical frameworks. Marktoberdorf Summer School Lectures, available online at `http://www.cs.cmu.edu/~fp/talks/mdorf01-slides.pdf`, July 2001.

[97] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science Publishers, 2001.

[98] Frank Pfenning and Carsten Schürmann. Algorithms for equality and unification in the presence of notational definitions. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs*, pages 179–193, Kloster Irsee, Germany, March 1998. Springer-Verlag LNCS 1657.

[99] Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In Harald Ganzinger, editor, *CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999*, volume 1632 of *Lecture Notes in Computer Science*, pages 202–206, 1999.

[100] Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL 98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 252–265, New York, NY, January 1998. ACM.

[101] Lutz Plümer. *Termination Proofs for Logic Programs*, volume 446 of *LNAI*. Springer-Verlag, 1990.

[102] Randy Pollack. Implicit Syntax. In Gérard Huet and Gordon Plotkin, editors, *Informal Proceedings of the 1st Workshop on Logical Frameworks (LF'90), Antibes*, pages 421–433, 1990.

[103] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[104] P. Sreeranga Rajan. Executing HOL specifications: Towards an evaluation semantics for classical higher order logic. In L. J. M. Claesen and M. J. C. Gordon, editors, *Higher order Logic Theorem Proving and its Applications*, Leuven, Belgium, September 1992. Elsevier.

[105] Claudio Sacerdoti Coen. Tactics in modern proof-assistants: The bad habit of overkilling. In Richard J. Boulton and Paul B. Jackson, editors, *TPHOLs 2001: Supplemental Proceedings*, number EDI-INF-RR-0046 in Informatics Report Series, pages 352–367. Division of Informatics, University of Edinburgh, Edinburgh, Scotland, UK, September 2001.

[106] Donald Sannella and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica*, 25:233–281, 1988.

[107] Donald Sannella and Martin Wirsing. A kernel language for algebraic specification and implementation. In *Proc. 1983 Intl. Conf. on Foundations of Computation Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 1983.

[108] Monika Seisenberger. Konstruktive Aspekte von Higmans Lemma. Master's thesis, Fakultät für Mathematik, Ludwig-Maximilians-Universität München, 1998.

[109] Monika Seisenberger. *On the Constructive Content of Proofs*. PhD thesis, Fakultät für Mathematik, Ludwig-Maximilians-Universität München, 2003.

[110] Paula Severi and Nora Szasz. Studies of a theory of specifications with built-in program extraction. *Journal of Automated Reasoning*, 27 (1):61–87, July 2001.

[111] Konrad Slind. *Reasoning about Terminating Functional Programs.* PhD thesis, Institut für Informatik, TU München, 1999.

[112] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference, Glenelg, Australia*, pages 499–512, 1995.

[113] Robert F. Stärk. Input/output dependencies of normal logic programs. *Journal of Logic and Computation*, 4(3):249–262, 1994.

[114] Aaron Stump. *Checking Validities and Proofs with CVC and flea.* PhD thesis, Stanford University, August 2002.

[115] William W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.

[116] Anne S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics, Volume 1*, volume 121 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam, 1988.

[117] Joakim von Wright. Representing higher-order logic proofs in HOL. In Thomas F. Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications, 7th International Workshop*, volume 859 of *Lecture Notes in Computer Science*, pages 456–470. Springer-Verlag, 1994.

[118] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobalt, and Dalibor Topic. SPASS version 2.0. In Andrei Voronkov, editor, *Automated Deduction – CADE-18*, volume 2392 of *Lecture Notes in Computer Science*, pages 275–279. Springer-Verlag, July 27-30 2002.

[119] Markus Wenzel. Type classes and overloading in higher-order logic. In Elsa L. Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: TPHOLs'97*, LNCS 1275, 1997.

[120] Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents.* PhD thesis, Institut für Informatik, TU München, 2002. `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html`.

[121] Benjamin Werner. *Une Théorie des Constructions Inductives.* PhD thesis, Université Paris 7, 1994.

[122] Wai Wong. Recording and checking HOL proofs. In E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications. 8th International Workshop*, volume 971 of *LNCS*, pages 353–68. Springer-Verlag, Berlin, 1995.