Thesis for the degree of Doctor of Philosophy

Towards a practical programming language based on dependent type theory

Ulf Norell

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden, 2007

Towards a practical programming language based on dependent type theory
Ulf Norell

ISBN 978-91-7291-996-9
ISSN 0346-718X

Doktorsavhandlingar vid Chalmers tekniska högskola, Ny serie Nr 2677.

Technical report 33D
Department of Computer Science and Engineering
Research group: Programming Logic

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden

Telephone +46 (0)31-772 1000

Printed at the Department of Computer Science and Engineering
Göteborg, 2007

Abstract

Dependent type theories [ML72] have a long history of being used for theorem proving. One aspect of type theory which makes it very powerful as a proof language is that it mixes deduction with computation. This also makes type theory a good candidate for programming—the strength of the type system allows properties of programs to be stated and established, and the computational properties provide semantics for the programs.

This thesis is concerned with bridging the gap between the theoretical presentations of type theory and the requirements on a practical programming language. Although there are many challenging research problems left to solve before we have an industrial scale programming language based on type theory, this thesis takes us a good step along the way.

In functional programming languages pattern matching provides a concise notation for defining functions. In dependent type theory, pattern matching becomes even more powerful, in that inspecting the value of a particular term can reveal information about the types and values of other terms. In this thesis we give a type checking algorithm for definitions by pattern matching in type theory, supporting overlapping patterns, and pattern matching on intermediate results using the *with* rule [MM04a].

Traditional presentations of type theory suffers from rather verbose notation, cluttering programs and proofs with, for instance, explicit type information. One solution to this problem is to allow terms that can be inferred automatically to be omitted. This is usually implemented by inserting metavariables in place of the omitted terms and using unification to solve these metavariables during type checking. We present a type checking algorithm for a theory with metavariables and prove its soundness independent of whether the metavariables are solved or not.

In any programming language it is important to be able to structure large programs into separate units or modules and limit the interaction between these modules. In this thesis we present a simple, but powerful module system for a dependently typed language. The main focus of the module system is to manage the name space of a program, and an important characteristic is a clear separation between the module system and the type checker, making it largely independent of the underlying language.

As a side track, not directly related to the use of type theory for programming, we present a connection between type theory and a first-order logic theorem prover. This connection saves the user the burden of proving simple, but tedious first-order theorems by leaving them for the prover. We use a transparent translation to first-order logic which makes the proofs constructed by the theorem prover human readable. The soundness of the

connection is established by a general metatheorem.

Finally we put our work into practise in the implementation of a programming language, Agda, based on type theory. As an illustrating example we show how to program a simple certified prover for equations in a commutative monoid, which can be used internally in Agda. Much more impressive examples have been done by others, showing that the ideas developed in this thesis are viable in practise.

Acknowledgements

I would like to thank all the people without whom this thesis would not have been possible. My supervisor Patrik Jansson for supporting me throughout my studies, my colleagues Andreas Abel, Catarina Coquand, Thierry Coquand, Nils Anders Danielsson, Peter Dybjer, and many more for excellent collaborations and many fruitful discussions about my work, Conor McBride for providing invaluable feedback on the thesis, my fiancée Cecilia for keeping me sane during the process of writing the thesis, and last but not least, my mother for teaching me programming 20 some years ago.

Contents

Chapter 1

Introduction

Programming is the craft of giving instructions to machines. Being machines they will follow these instructions regardless of whether they make sense or not. The purpose of a programming language is to make it easier to express the things that do make sense while making it harder or impossible to express things that do not make sense.

The first step in this direction is to make programming languages readable by human beings. That way the programmer can read her program and convince herself that it makes sense, but since programmers are humans they will make mistakes both writing programs and trying to make sense of them. To help with this modern programming languages come equipped with type systems, which allows the programmer to declare the purpose of a program in the form of a *type*. The machine can then check that a given program has the intended behaviour. Types can also be used to guide the programmer in constructing the correct program.

In its simplest form a type system allows you to state, for instance, that the purpose of a sorting program is to take a list as input and produce a list as output. This is a very crude approximation of what it means to be a sorting program but it does provide some guarantees. In more expressive type systems, such as the ones discussed in this thesis, it is possible to express that the sorting program takes a list of elements over which there is a total order, and computes an ordered permutation of this list. This characterises exactly what it means to be a sorting program and so once we have written a program matching this specification we know that it is correct. There is a trade-off here: the more precise we make the types the more we might have to explain in order for the machine to see that the program matches our intentions. But more precise types also means that we can get more help from the machine when constructing a program.

This thesis deals with the problem of building a programming language

based on a dependent type theory in which very precise statements of the purpose of a program can be made. The main contributions are:

- An algorithm for type checking pattern matching equations over inductive families of datatypes,
- a type-safe treatment of metavariables, enabling a form of implicit syntax,
- a simple but powerful module system,
- a way to connect the type checker to a first-order logic theorem prover to allow simple proofs to be found automatically, and
- an implementation of a programming language, Agda, proving that practical programming with dependent types is within our reach.

1.1 Overview of the thesis

The rest of this chapter sets the scene by introducing a dependent type theory UTT_Σ and a type checking algorithm for this theory.

The following three chapters deal with the task of turning this basic theory into a programming language, adding pattern matching, metavariables, and a module system.

Chapter 2 discusses how to extend the theory with inductive families and functions defined by pattern matching over elements in these families. We give a type checking algorithm and an algorithm for checking coverage of pattern match definitions.

In Chapter 3 we describe and prove sound an algorithm for type checking a type theory extended with metavariables. This allows us to extend our language with a notion of implicit arguments.

Chapter 4 describes a simple but powerful module system for dependently typed languages. By keeping the module system separate from the type checker we obtain a clean module system which is largely independent of the underlying language.

The results from the these chapters are put to good use in the implementation of the Agda language, which is described from a user's perspective in Chapter 5. A bigger example of an Agda program for proving equations in a commutative monoid is given.

Chapter 6 digresses from the theme of using type theory for programming, and shows how a first-order logic theorem prover can be connected to a dependent type theory to provide automation of proofs of first-order theorems.

1.2 Context

Dependent type theories have been around since the early 1970's, when Martin-Löf introduced his intuitionistic theory of types [ML72]. The original motivation for type theory was to serve as a basis for constructive mathematics, and as such it has been very successful. Proof assistants such as Coq [BC04], NuPrl [CAB+86], Alf [MN94], Agda[1] [CC99], and Lego [Pol94] have made it possible to construct very impressive proofs in type theory and have them formally checked by a computer.

It is only in the last ten years that the interest in using type theory and dependent types for programming has grown stronger. This topic can be approached from two sides: taking a type theory and turning it into a programming language or starting with a programming language and adding type theoretic features to it.

The former approach was taken in the Cayenne [Aug98] language, where Martin-Löf type theory was combined with general recursion. This had the unfortunate side-effect of making type checking in Cayenne undecidable, since the type checker might have to evaluate arbitrary non-terminating expressions.

In his thesis [McB99], McBride extended Lego with facilities for interactive programming and pattern matching. These ideas were later refined by McBride and McKinna [MM04a] and led to the development of the Epigram language [McB07]. The programming model of Epigram is very similar to what we present in this thesis and has been a great inspiration. Unfortunately, the implementation of Epigram has not yet reached a level where it can be used for writing bigger programs.

Another interesting recent development is the Delphin language by Poswolsky and Schürmann [PS07]. Delphin is a dependently typed programming language built on top of the LF logical framework [HHP93]. The main focus of Delphin is on manipulating higher-order syntax, something which is made easy by the introduction of a *newness* operator, allowing the quantification over fresh constants. Delphin supports pattern matching over inductive families, but where we, in our work, get away with first order matching, Delphin uses unification and higher-order matching at run-time.

Recently there has been some work on using Coq as a programming language. Impressive certified programs have been written by, among others, Chlipala [Chl06, Chl07], and Leroy [Ler06]. Furthermore, Sozeau has developed Russell [Soz07], a layer on top of Coq to make programming easier. The

[1] This refers to the predecessor of the language developed in this thesis, which is also called Agda.

idea is that one writes dependently typed programs as if they were simply typed. The proof obligations arising from the dependent types are recorded by Russell and can be proved separately using the tactic language of Coq. This approach is quite appealing in that it separates the program logic from the proofs required to show well-typedness of the program.

There has also been a lot of work from the other direction—adding dependent types to conventional programming languages. Dependent ML [Xi98] extends ML with types dependent on integers, and Haskell has recently been extended with generalised algebraic datatypes (GADT) [PVWW06], a restricted form of inductive families. There has also been some new languages, such as Applied Type Systems [Xi04] and Omega [She05]. Common for these languages and extensions is that they only support a limited form of type dependencies. For instance, there is no way of having a type depending on the value of another dependent type.

1.3 A basic dependent type theory

What sets dependent type theory apart from other type theories is that types can depend on terms. In a non-dependent theory types and terms live in separate worlds and they only meet to decide what terms have which types. In a dependent theory, on the other hand, types can talk about terms and so it is possible to express things like the precise characterisation of the sorting function mentioned above.

In this section we present a dependent type theory which can serve as basis for the extensions discussed in later chapters. The particular choice of type theory is not crucial and the theory we choose is roughly Luo's UTT [Luo94] extended with Σ-types and η-laws. In the following we will refer to this theory as UTT_Σ. The syntax of UTT_Σ is presented in Figure 1.1.

A telescope [dB91b] $\Delta = (x_1 : A_1) \dots (x_n : A_n)$ is a sequence of types where later types may depend on elements of previous types. When there are consecutive occurrences of a type in a telescope we may combine them and write, for instance, $(x\ y\ :\ A)(z\ :\ B)$ for $(x\ :\ A)(y\ :A)(z\ :\ B)$.

Dependent type theory generalises the simple function space $A \to B$ to a dependent function space $(x : A) \to B$ where the result type B can depend on the value of the argument. We sometimes refer to dependent function types as Π-types for mathematical reasons. If B does not depend on x we allow ourselves to write $A \to B$ and if we have a telescope $\Delta = (x_1 : A_1) \dots (x_n : A_n)$ we write $\Delta \to B$ for $(x_1 : A_1) \to \dots \to (x_n : A_n) \to B$. Functions are introduced by λ-terms $\lambda x.\ t$ and computes by β-reduction. To abstract over a sequence of variables $\bar{x}$ we write $\lambda \bar{x}.\ t$ or $\lambda \Delta.\ t$ rather than $\lambda x_1 \dots \lambda x_n.\ t$.

$s,\ t,\ A,\ B$	$::=$	x	variable
	$\mid$	$(x\ :\ A)\ \to\ B$	dependent function type
	$\mid$	$\lambda x.\ t$	lambda abstraction
	$\mid$	$s\ t$	function application
	$\mid$	$(x\ :\ A)\ \times\ B$	dependent pair type
	$\mid$	$\langle s,\ t\rangle$	dependent pairs
	$\mid$	$\pi_1\ t\ \mid\ \pi_2\ t$	projection
	$\mid$	Set_i	universes ($i \in \{0..\}$)
	$\mid$	1	the unit type
	$\mid$	$\langle\rangle$	the element of the unit type
$\Gamma,\ \Delta$	$::=$	ε	
	$\mid$	$(x\ :\ A)\Gamma$	telescopes

Figure 1.1: The syntax of UTT_Σ

Function application is represented by juxtaposition and analogously with λ-abstraction we write $t\ \bar{x}$ or $t\ \Delta$ for $t\,x_1\ \ldots\ x_n$.

Similarly to function types the product type $A \times B$ generalises to the type of dependent pairs $(x : A) \times B$ where the type of the second component depends on the value of the first component. We call a dependent pair type a Σ-type. The elements are constructed and deconstructed as usual. We write $\langle s, t\rangle$ for the dependent pair of s and t, and $\pi_1\ t$ and $\pi_2\ t$ to project the first and second components, respectively, from a pair t. The computation rules are the expected ones. We also include a singleton type $\mathsf{1}$ with the single element $\langle\rangle$.

Types inhabit a cumulative hierarchy of universes Set_i each closed under Π and Σ, and included in the next higher universe. Lower universes are also embedded in higher universes by the subtyping relation. We omit the level index for the smallest universe and write Set for Set_0. There are many different ways of adding a hierarchy of universes [ML75, ML84] and the precise way it is done is not crucial to this thesis.

We write $t[x := s]$ for the usual, capture avoiding, substitution of s for x in t. For the simultaneous substitution of a sequence of terms we write $t[\bar{x} := \bar{s}]$ or $t[\Delta := \bar{s}]$.

The typing rules are presented in Figure 1.2 and the conversion rules in Figure 1.3. We ignore variable freshness conditions in the rules. In practise, these can be handled by adopting a suitable discipline, such as de Bruijn indices [Bru72]. The typing rules expresses the expected relation between a term and its type. Of particular interest is the subtyping rule which states

Contexts: $\boxed{\Gamma \vdash \textbf{valid}}$

$$\frac{}{\vdash \textbf{valid}} \qquad \frac{\Gamma \vdash \textbf{valid} \qquad \Gamma \vdash A : \mathsf{Set}_i}{\Gamma, x : A \vdash \textbf{valid}}$$

Types and terms: $\boxed{\Gamma \vdash t : A}$

$$\frac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash \mathsf{Set}_i : \mathsf{Set}_{i+1}} \qquad \frac{\Gamma \vdash A : \mathsf{Set}_i \qquad \Gamma, x : A \vdash B : \mathsf{Set}_i}{\Gamma \vdash (x : A) \times B : \mathsf{Set}_i}$$

$$\frac{\Gamma \vdash A : \mathsf{Set}_i \qquad \Gamma, x : A \vdash B : \mathsf{Set}_i}{\Gamma \vdash (x : A) \to B : \mathsf{Set}_i} \qquad \frac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash 1 : \mathsf{Set}_0} \qquad \frac{\Gamma \vdash \textbf{valid} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B[x := s]}{\Gamma \vdash \langle s, t \rangle : (x : A) \times B} \qquad \frac{\Gamma \vdash t : (x : A) \times B}{\Gamma \vdash \pi_1\ t : A} \qquad \frac{\Gamma \vdash t : (x : A) \times B}{\Gamma \vdash \pi_2\ t : B[x := \pi_1\ t]}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : (x : A) \to B} \qquad \frac{\Gamma \vdash s : (x : A) \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\ t : B[x := t]} \qquad \frac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash \langle\rangle : 1}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \leqslant B}{\Gamma \vdash t : B}$$

Figure 1.2: Typing rules for UTT_Σ

Subtyping: $\boxed{\Gamma \vdash A \leqslant B}$

$$\frac{}{\Gamma \vdash \mathsf{Set}_i \leqslant \mathsf{Set}_{i+1}}$$

$$\frac{\Gamma \vdash A_1 : \mathsf{Set}_i \qquad \Gamma \vdash A_2 : \mathsf{Set}_i \qquad \Gamma \vdash A_1 \simeq A_2 : \mathsf{Set}_i \qquad \Gamma, x : A_1 \vdash B_1 \leqslant B_2}{\Gamma \vdash (x : A_1) \to B_1 \leqslant (x : A_2) \to B_2}$$

$$\frac{\Gamma \vdash A_1 : \mathsf{Set}_i \qquad \Gamma \vdash A_2 : \mathsf{Set}_i \qquad \Gamma \vdash A_1 \simeq A_2 : \mathsf{Set}_i \qquad \Gamma, x : A_1 \vdash B_1 \leqslant B_2}{\Gamma \vdash (x : A_1) \times B_1 \leqslant (x : A_2) \times B_2}$$

$$\frac{\Gamma \vdash A_1 : \mathsf{Set}_i \qquad \Gamma \vdash A_2 : \mathsf{Set}_i \qquad \Gamma \vdash A_1 \simeq A_2 : \mathsf{Set}_i}{\Gamma \vdash A_1 \leqslant A_2}$$

$$\frac{\Gamma \vdash A_1 \leqslant A_2 \qquad \Gamma \vdash A_2 \leqslant A_3}{\Gamma \vdash A_1 \leqslant A_3}$$

Reduction: $\boxed{s \to_\beta t}$

$$\frac{}{(\lambda x.s)\ t \to_\beta s[x := t]} \qquad \frac{}{\pi_1\ \langle s, t\rangle \to_\beta s} \qquad \frac{}{\pi_2\ \langle s, t\rangle \to_\beta t}$$

Conversion: $\boxed{\Gamma \vdash s \simeq t : A}$

$$\frac{}{\Gamma \vdash t \simeq \lambda x.\, t\, x : (x : A) \to B} \qquad \frac{}{\Gamma \vdash t \simeq \langle \pi_1\ t, \pi_2\ t\rangle : (x : A) \times B}$$

$$\frac{}{\Gamma \vdash t \simeq \langle\rangle : 1} \qquad \frac{s \to_\beta t}{\Gamma \vdash s \simeq t : A} \qquad \frac{}{\Gamma \vdash t \simeq t : A} \qquad \frac{\Gamma \vdash t \simeq s : A}{\Gamma \vdash s \simeq t : A}$$

$$\frac{\Gamma \vdash t_1 \simeq t_2 : A \qquad \Gamma \vdash t_2 \simeq t_3 : A}{\Gamma \vdash t_1 \simeq t_3 : A} \qquad + \textit{congruences}$$

Figure 1.3: Conversion rules for UTT_Σ

that if t has type A and A is a subtype of B then t has type B. The subtyping relation is the extension of the fact that Set_i is a subtype of Set_j if $i \leqslant j$. We have chosen Σ and Π to be invariant in their first argument, but it is also conceivable to make them covariant and contravariant, respectively. The conversion rules implement $\beta\eta$-equality on terms. Worth noting is that β-equality is represented by a reduction relation, whereas η-equality is judgemental. This presentation corresponds to how conversion is implemented in the type checking rules in the next section.

In the current presentation we cannot write very many interesting programs since the only base type we have is the singleton type. Rather than adding more interesting base types, however, we hold out until Chapter 2 where we show how to add inductively defined families of types [Dyb94]. For now we make do with the examples of the polymorphic identity function and a dependent function composition.

$$
\begin{aligned}
&\mathit{id} \;:\; (A \;:\; \mathsf{Set}) \;\to\; A \;\to\; A \\
&\mathit{id} \;=\; \lambda A\; x.\; x
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{comp} \;:\; (A\; B \;:\; \mathsf{Set})(C \;:\; B \;\to\; \mathsf{Set}) \;\to \\
&\qquad ((x \;:\; B) \;\to\; C\; x) \;\to\; (g \;:\; A \;\to\; B)(x \;:\; A) \;\to\; C\; (g\; x) \\
&\mathit{comp} \;=\; \lambda A\; B\; C\; f\; g\; x.\; f\; (g\; x)
\end{aligned}
$$

This composition operator is not the most general possible—we could also make g a dependent function—but it is sufficiently general for most common applications. It also has the nice property that the type arguments A, B, and C, can be inferred automatically (see Chapter 3).

1.4 Type checking

We now present a type checking algorithm for UTT_Σ. We use a bidirectional algorithm with mutually defined judgements for checking an expression against a type and inferring the type of an expression [Pau90, Coq96]. We also let the type checker produce a well-typed term from the input expression rather than just check that it is well-typed, thus separating the user language from the core language of the type checker. These two languages have distinctly different purposes—the user language should be friendly to the user, whereas the core language should be friendly to the type checker. For instance, the user language might use named variables whereas for the core language we may want to handle names using de Bruijn indices or de Bruijn levels, or a combination of both [MM04b]. Furthermore, when we

Type checking: $\boxed{\Gamma \vdash e \uparrow A \rightsquigarrow t}$

$$\frac{A \rightarrow_{whnf} (x : B) \to C \qquad \Gamma, x : B \vdash e \uparrow C \rightsquigarrow t}{\Gamma \vdash \lambda x.\ e \uparrow A \rightsquigarrow \lambda x.\ t}$$

$$\frac{A \rightarrow_{whnf} (x : B) \times C \qquad \Gamma \vdash e_1 \uparrow B \rightsquigarrow s \qquad \Gamma \vdash e_2 \uparrow C[x := s] \rightsquigarrow t}{\Gamma \vdash \langle e_1, e_2 \rangle \uparrow A \rightsquigarrow \langle s, t \rangle}$$

$$\frac{\Gamma \vdash e \downarrow B \rightsquigarrow t \qquad \Gamma \vdash A \leqslant B}{\Gamma \vdash e \uparrow A \rightsquigarrow t}$$

Figure 1.4: Type checking rules.

add metavariables in Chapter 3 the well-typed term constructed by the type checker will only be an approximation of the term given by the user.

We end up with the following two judgements for type checking and type inference:

$\Gamma \vdash e \downarrow A \rightsquigarrow t$	Inferring the type of e in the context Γ
$\Gamma \vdash e \uparrow A \rightsquigarrow t$	Checking that e has type A in the context Γ

The intuition behind the up and down arrows is that when checking, the type is pushed upwards in the derivation tree, whereas during inference the type is computed from the leaves of the tree. In other words, when checking the inputs are Γ, e and A, and the output is t. During inference the inputs are Γ and e and the outputs A and t. The rules maintain the invariant that $\Gamma \vdash A : \mathsf{Set}_i$ for some i, which in turn implies $\Gamma \vdash \mathbf{valid}$. Soundness of the type checker (which we do not prove here) gives $\Gamma \vdash t : A$.

The syntax directed rules for type checking and inference are given in Figure 1.4 and Figure 1.5. We require the type to be available when checking λ-abstractions and pairs. In the case of a λ-abstraction we do not know the type that is abstracted over and in the case of a dependent pair we cannot infer how the type of the second component depends on the value of the first. A consequence of this is that we cannot type check β-redexes. This is not a severe limitation in practise, but it does mean that any completeness results of the algorithm have to be stated relative to β-normal terms.

Types can be arbitrary terms which might not be in normal form. For instance, when checking the type of a λ-function we cannot demand that

Type inference: $\boxed{\Gamma \vdash e \downarrow A \rightsquigarrow t}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \downarrow A \rightsquigarrow x} \qquad \frac{}{\Gamma \vdash \langle\rangle \downarrow 1 \rightsquigarrow \langle\rangle}$$

$$\frac{\Gamma \vdash e_1 \downarrow A \rightsquigarrow s \qquad A \rightarrow_{whnf} (x : B) \rightarrow C \qquad \Gamma \vdash e_2 \uparrow B \rightsquigarrow t}{\Gamma \vdash e_1\ e_2 \downarrow C[x := t] \rightsquigarrow s\ t}$$

$$\frac{\Gamma \vdash e \downarrow A \rightsquigarrow t \qquad A \rightarrow_{whnf} (x : B) \times C}{\Gamma \vdash \pi_1\ e \downarrow B \rightsquigarrow \pi_1\ t}$$

$$\frac{\Gamma \vdash e \downarrow A \rightsquigarrow t \qquad A \rightarrow_{whnf} (x : B) \times C}{\Gamma \vdash \pi_2\ e \downarrow C[x := \pi_1\ t] \rightsquigarrow \pi_2\ t}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \downarrow C_1 \rightsquigarrow A \qquad \Gamma, x : A \vdash e_1 \downarrow C_2 \rightsquigarrow B \\ C_1 \rightarrow_{whnf} \mathsf{Set}_i \qquad C_2 \rightarrow_{whnf} \mathsf{Set}_j\end{array}}{\Gamma \vdash (x : e_1) \rightarrow e_2 \downarrow \mathsf{Set}_{i \sqcup j} \rightsquigarrow (x : A) \rightarrow B}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \downarrow C_1 \rightsquigarrow A \qquad \Gamma, x : A \vdash e_1 \downarrow C_2 \rightsquigarrow B \\ C_2 \rightarrow_{whnf} \mathsf{Set}_i \qquad C_1 \rightarrow_{whnf} \mathsf{Set}_j\end{array}}{\Gamma \vdash (x : e_1) \times e_2 \downarrow \mathsf{Set}_{i \sqcup j} \rightsquigarrow (x : A) \times B}$$

$$\frac{}{\Gamma \vdash 1 \downarrow \mathsf{Set}_0 \rightsquigarrow 1} \qquad \frac{}{\Gamma \vdash \mathsf{Set}_i \downarrow \mathsf{Set}_{i+1} \rightsquigarrow \mathsf{Set}_i}$$

Figure 1.5: Type inference rules.

Subtyping: $\boxed{\Gamma \vdash A \leqslant B}$

$$\frac{A \to_{whnf} A' \qquad B \to_{whnf} B' \qquad \Gamma \vdash A' \leqslant' B'}{\Gamma \vdash A \leqslant B}$$

Subtyping (weak head normal forms): $\boxed{\Gamma \vdash A \leqslant' B}$

$$\frac{}{\Gamma \vdash \mathsf{Set}_i \leqslant' \mathsf{Set}_{i+1}} \qquad \frac{\Gamma \vdash A_1 \simeq A_2 \uparrow \mathsf{Set}_\alpha \qquad \Gamma, x : A_1 \vdash B_1 \leqslant B_2}{\Gamma \vdash (x : A_1) \to B_1 \leqslant' (x : A_2) \to B_2}$$

$$\frac{\Gamma \vdash A_1 \simeq A_2 \uparrow \mathsf{Set}_\alpha \qquad \Gamma, x : A_1 \vdash B_1 \leqslant B_2}{\Gamma \vdash (x : A_1) \times B_1 \leqslant' (x : A_2) \times B_2} \qquad \frac{\Gamma \vdash A \simeq' B \uparrow \mathsf{Set}_\alpha}{\Gamma \vdash A \leqslant' B}$$

Figure 1.6: Subtype checking.

the type *is* a Π-type, merely that is *computes* to a Π-type. We denote by $t \to_{whnf} nf$ the reduction of the term t to its weak head normal form nf defined using the rules for β-reduction from Figure 1.3. Weak head normal forms are described by the following grammar:

$$\begin{array}{rcl} nf & ::= & ne \mid \lambda x.\, t \mid \langle s,\, t \rangle \mid \langle\rangle \\ & \mid & (x : A) \to B \mid (x : A) \times B \mid 1 \mid \mathsf{Set}_i \\ ne & ::= & ne\ s \mid \pi_1\ ne \mid \pi_2\ ne \end{array}$$

We do not use special names for normal and neutral terms in the following, but continue using s and t for all forms of terms.

If in checking mode, we encounter a term for which we can infer the type, we do so and check that the inferred type is a subtype of the expected type. The type inference rules are very similar to the typing rules from Figure 1.2. Notable differences are the explicit computation of weak head normal forms and the computation of the universe level of Σ and Π-types (we write $i \sqcup j$ for the maximum of i and j).

When checking subtyping (Figure 1.6) the two types are first (weak head) normalised.

$\Gamma \vdash A \leqslant B$ checking subtyping between arbitrary types
$\Gamma \vdash A \leqslant' B$ checking subtyping between normal types

Conversion: $\boxed{\Gamma \vdash s \simeq t \uparrow A}$

$$\frac{s \rightarrow_{whnf} s' \qquad t \rightarrow_{whnf} t' \qquad A \rightarrow_{whnf} A' \qquad \Gamma \vdash s' \simeq' t' \uparrow A'}{\Gamma \vdash s \simeq t \uparrow A}$$

Conversion (weak head normal forms): $\boxed{\Gamma \vdash s \simeq' t \uparrow A}$

$$\frac{}{\Gamma \vdash \mathsf{Set}_i \simeq' \mathsf{Set}_i \uparrow \mathsf{Set}_\alpha} \qquad \frac{\Gamma \vdash A_1 \simeq A_2 \uparrow \mathsf{Set}_\alpha \qquad \Gamma, x : A_1 \vdash B_1 \simeq B_2 \uparrow \mathsf{Set}_\alpha}{\Gamma \vdash (x : A_1) \to B_1 \simeq' (x : A_2) \to B_2 \uparrow \mathsf{Set}_\alpha}$$

$$\frac{\Gamma \vdash A_1 \simeq A_2 \uparrow \mathsf{Set}_\alpha \qquad \Gamma, x : A_1 \vdash B_1 \simeq B_2 \uparrow \mathsf{Set}_\alpha}{\Gamma \vdash (x : A_1) \times B_1 \simeq' (x : A_2) \times B_2 \uparrow \mathsf{Set}_\alpha} \qquad \frac{}{\Gamma \vdash s \simeq' t \uparrow 1}$$

$$\frac{\Gamma, x : A \vdash s\ x \simeq t\ x \uparrow B}{\Gamma \vdash s \simeq' t \uparrow (x : A) \to B}$$

$$\frac{\Gamma \vdash \pi_1\ s \simeq \pi_1\ t \uparrow A \qquad \Gamma \vdash \pi_2\ s \simeq \pi_2\ t \uparrow B[x := \pi_1\ s]}{\Gamma \vdash s \simeq' t \uparrow (x : A) \times B}$$

$$\frac{s, t\ \mathit{neutral} \qquad \Gamma \vdash s \equiv t \downarrow A'}{\Gamma \vdash s \simeq' t \uparrow A}$$

Figure 1.7: Type directed conversion checking

Equality of neutral terms: $\boxed{\Gamma \vdash s \equiv t \downarrow A}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \equiv x \downarrow A} \qquad \frac{\Gamma \vdash s_1 \equiv s_2 \downarrow A \qquad A \rightarrow_{whnf} (x : B) \rightarrow C \qquad \Gamma \vdash t_1 \simeq t_2 \uparrow B}{\Gamma \vdash s_1\ t_1 \equiv s_2\ t_2 \downarrow C[x := t_1]}$$

$$\frac{\Gamma \vdash s \equiv t \downarrow A \qquad A \rightarrow_{whnf} (x : B) \times C}{\Gamma \vdash \pi_1\ s \equiv \pi_1\ t \downarrow B}$$

$$\frac{\Gamma \vdash s \equiv t \downarrow A \qquad A \rightarrow_{whnf} (x : B) \times C}{\Gamma \vdash \pi_2\ s \equiv \pi_2\ t \downarrow C[x := \pi_1\ s]}$$

Figure 1.8: Conversion checking for neutral terms.

In these rules we assume the invariant that $\Gamma \vdash A : \mathsf{Set}_i$ and $\Gamma \vdash B : \mathsf{Set}_j$ for some i and j.

The same is done for convertibility (Figure 1.7, but here we make a further distinction between conversion checking neutral terms and terms in weak head normal form:

$\Gamma \vdash s \simeq t \uparrow A$	checking conversion of arbitrary terms
$\Gamma \vdash s \simeq' t \uparrow A$	checking conversion of normal terms
$\Gamma \vdash s \equiv t \downarrow A$	checking conversion of neutral terms

Analogously to the rules for type checking and inference, we use a bidirectional approach to conversion checking, inferring the type when comparing neutral terms. The conversion checking of arbitrary terms uses the type to guide η-expansion, thus, when switching from checking subtyping to checking conversion we have to recover the types. When the type is a sort (Set_i) the particular level does not matter—it does not affect η-conversions—so we write Set_α for an arbitrary sort. In principle we could recover the level, but it is not necessary. For Π and Σ, η-conversion can also be done in an untyped way [AC05], but this approach breaks down once we have the η-law for $\mathsf{1}$.

The invariants for the conversion checking rules are $\Gamma \vdash s : A$ and $\Gamma \vdash t : A$ [2]. Note that when switching between neutral and normal terms there is no need to check that the inferred type corresponds to the given type.

[2] In the case when $A = \mathsf{Set}_\alpha$ this means that there exists an i such that $\Gamma \vdash s : \mathsf{Set}_i$ and $\Gamma \vdash t : \mathsf{Set}_i$.

1.5 Extensions to the theory

In the coming chapters we will discuss various extensions to UTT_Σ. To prepare the reader we outline these extensions here.

1.5.1 Inductive definitions

In Chapter 2 we describe a type checking algorithm for definitions by pattern matching over inductively defined families of datatypes. A datatype family is introduced by a **data** declaration:

$$\begin{array}{l}\textbf{data}\ D\ \Delta\ :\ \Gamma\ \rightarrow\ \mathsf{Set}_i\ \textbf{where}\\ \quad \mathsf{c}_1\ :\ \Theta_1\ \rightarrow\ D\ \Delta\ \bar{t_1}\\ \quad \vdots\\ \quad \mathsf{c_n}\ :\ \Theta_n\ \rightarrow\ D\ \Delta\ \bar{t_n}\end{array}$$

This declaration introduces a datatype family D indexed over Γ and parameterised by Δ, inductively defined by the constructors $\mathsf{c_1} \ldots \mathsf{c_n}$ with the given types. The parameters Δ scope over the types of the constructors and must be unchanged in the targets of the constructors, whereas each constructor can target a different index. For ordinary non-family datatypes Γ will be empty. For instance, the datatype of natural numbers can be introduced by

$$\begin{array}{l}\textbf{data}\ \mathit{Nat}\ :\ \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{zero}\ :\ \mathit{Nat}\\ \quad \mathsf{suc}\ \ :\ \mathit{Nat}\ \rightarrow\ \mathit{Nat}\end{array}$$

and the family of n-element finite sets is given by

$$\begin{array}{l}\textbf{data}\ \mathit{Fin}\ :\ \mathit{Nat}\ \rightarrow\ \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{fzero}\ :\ (n\ :\ \mathit{Nat})\ \rightarrow\ \mathit{Fin}\ (\mathsf{suc}\ n)\\ \quad \mathsf{fsuc}\ \ :\ (n\ :\ \mathit{Nat})\ \rightarrow\ \mathit{Fin}\ n\ \rightarrow\ \mathit{Fin}\ (\mathsf{suc}\ n)\end{array}$$

An example of a parameterised datatype is the type of lists over a set A.

$$\begin{array}{l}\textbf{data}\ \mathit{List}\ (A\ :\ \mathsf{Set})\ :\ \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{nil}\ \ \ :\ \mathit{List}\ A\\ \quad \mathsf{cons}\ :\ A\ \rightarrow\ \mathit{List}\ A\ \rightarrow\ \mathit{List}\ A\end{array}$$

If we index the lists by their length we get the family of vectors:

$$\begin{array}{l}\textbf{data}\ \mathit{Vec}\ (A\ :\ \mathsf{Set})\ :\ \mathit{Nat}\ \rightarrow\ \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{vnil}\ \ \ :\ \mathit{Vec}\ A\ \mathsf{zero}\\ \quad \mathsf{vcons}\ :\ (n\ :\ \mathit{Nat})\ \rightarrow\ A\ \rightarrow\ \mathit{Vec}\ A\ n\ \rightarrow\ \mathit{Vec}\ A\ (\mathsf{suc}\ n)\end{array}$$

1.5.2 Uniqueness of identity proofs

An inductive family which is particularly interesting is the identity type, which can be defined by

$$\begin{array}{l}\textbf{data}\ Id\ (A\ :\ \mathsf{Set})(x\ :\ A)\ :\ A\ \to\ \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{refl}\ :\ Id\ A\ x\ x\end{array}$$

For any $(A : \mathsf{Set})(x : A)$ we have a family of datatypes indexed over A, which is empty at all indices except x. The situation at index x is not entirely straightforward. The axiom K introduced by Streicher [Str93] implies that refl is the unique element of type $Id\ A\ x\ x$[3]:

$$\begin{array}{l}K\ :\ (A\ :\ Set)(x\ :\ A)(P\ :\ Id\ A\ x\ x\ \to\ \mathsf{Set})\ \to\\ \qquad P\ \mathsf{refl}\ \to\ (p\ :\ Id\ A\ x\ x)\ \to\ P\ p\end{array}$$

It has been shown by Hofmann and Streicher [HS94] that this axiom is not derivable from the elimination rule for Id. However, in the presence of definitions by pattern matching one would expect this axiom to hold. An entirely plausible definition of K can be obtained by pattern matching on p:

$$K\ A\ x\ P\ pr\ \mathsf{refl}\ =\ pr$$

In fact McBride [McB99, MM04a, GMM06] has shown that this is the only axiom in addition to the standard elimination rules that is needed to represent definitions by pattern matching in type theory.

1.5.3 Record types

It is straightforward to use Σ-types to encode labelled record types. We declare a record in a similar way to datatypes, but instead of a sequence of constructors we list the record fields and their types. For instance,

$$\begin{array}{l}\textbf{record}\ R\ :\ \mathsf{Set}\ \textbf{where}\\ \quad x\ :\ A\\ \quad y\ :\ B\ x\\ \quad z\ :\ C\ x\ y\end{array}$$

for some $(A : \mathsf{Set})(B : A \to \mathsf{Set})(C : (x : A) \to B\ x \to \mathsf{Set})$. We can encode this type in UTT_Σ as

[3]The original statement of the K axiom was for the identity where both elements are indices [ML75]. The presentation given here is for the equivalent identity type, due to Paulin-Mohring [PPM90], indexed only over the second element. Using the latter simplifies the statement of the axiom somewhat.

$$R \;=\; (x \;:\; A) \;\times\; (y \;:\; B\,x) \;\times\; C\,x\,y$$

and field projection functions can be defined using the Σ-projections:

$$\begin{aligned}
&x \;:\; R \;\to\; A\\
&x\,r \;=\; \pi_1\,r\\
&y \;:\; (r \;:\; R) \;\to\; B\,(x\,r)\\
&y\,r \;=\; \pi_1\,(\pi_2\,r)\\
&z \;:\; (r \;:\; R) \;\to\; C\,(x\,r)\,(y\,r)\\
&z\,r \;=\; \pi_2\,(\pi_2\,r)
\end{aligned}$$

In practise, however, it is a good idea to let each record declaration introduce a new type. This means that two record types declared to have the same fields will be different, but they will have the same elements. One advantage of this is that it significantly improves the efficiency of checking equality between record types—instead of comparing the types of all the fields, it is enough to compare the names. It is also good programming practise to keep intentionally different types separate in the type system.

1.5.4 Implicit arguments

In Chapter 3 we give an algorithm for type checking in the presence of metavariables. This will allow us to extend our theory with implicit arguments. We introduce a new function space $\{x \;:\; A\} \;\to\; B$, semantically equivalent to $(x \;:\; A) \;\to\; B$ but where the argument can be omitted. For instance, the polymorphic identity function can be given the type

$$id \;:\; \{A \;:\; \mathsf{Set}\} \;\to\; A \;\to\; A$$

To apply the identity function to an element x of a type A, one simply writes $id\,x$, omitting the first argument. We will not impose any restrictions on where implicit function spaces are allowed, but rather report an error if the implicit arguments cannot be inferred in a particular instance. The reason for this is that it is not clear exactly what such restrictions would look like and they would necessarily exclude many useful cases of implicit arguments.

Chapter 2

Pattern Matching

In a simply typed setting pattern matching is a convenient mechanism for analysing the structure of values, and it is one of the strong points of popular functional languages such as ML and Haskell. In the presence of dependent types the scrutinee of a pattern match may appear in the goal type. Hence, pattern matching will instantiate the goal with the different patterns. When we introduce inductively defined families of datatypes [Dyb94], pattern matching becomes even more powerful. Consider, for instance, the simple datatype of natural numbers *Nat* and its inductively defined ordering relation $\_\leqslant\_$[1]:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Nat}\ :\ \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{zero}\ :\ \mathit{Nat} \\
\quad \mathsf{suc}\ \ :\ \mathit{Nat}\ \rightarrow\ \mathit{Nat} \\
\textbf{data}\ \_\leqslant\_\ :\ \mathit{Nat}\ \rightarrow\ \mathit{Nat}\ \rightarrow\ \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{leqZero}\ :\ (n\ :\ \mathit{Nat})\ \rightarrow\ \mathsf{zero}\ \leqslant\ n \\
\quad \mathsf{leqSuc}\ \ :\ (n\ m\ :\ \mathit{Nat})\ \rightarrow\ n\ \leqslant\ m\ \rightarrow\ \mathsf{suc}\ n\ \leqslant\ \mathsf{suc}\ m
\end{array}
$$

The major source of difficulty when moving from simply typed pattern matching to pattern matching over inductive families is that pattern matching on one value yields information about other values. This makes case-expressions unsuitable for pattern matching. In the example of the types above, given an element $p\ :\ n\ \leqslant m$ for some n and m, when pattern matching on p, n and m will be instantiated. In other words, when pattern matching on elements of a family, not only the goal type is instantiated, but also the context. Consider the problem of proving transitivity of $\leqslant$:

$$\mathit{trans}\ :\ (k\ m\ n\ :\ \mathit{Nat})\ \rightarrow\ k\ \leqslant\ m\ \rightarrow\ m\ \leqslant\ n\ \rightarrow\ k\ \leqslant\ n$$

[1] Names containing underscores can be used as operators where the arguments go in place of the underscores. Hence, $x \leqslant y$ is equivalent to $\_\leqslant\_\ x\ y$.

$$trans\ k\ m\ n\ km\ mn\ = ?$$

If we decide to pattern match on the proof of $k \leqslant m$ the problem is refined to

$$\begin{array}{llll} trans\ \mathsf{zero} & m & n\ (\mathsf{leqZero}\ m) & mn\ = ? \\ trans\ (\mathsf{suc}\ k)\ (\mathsf{suc}\ m) & n\ (\mathsf{leqSuc}\ k\ m\ km) & mn\ = ? \end{array}$$

We can close the first case with $\mathsf{leqZero}\ n$ and in the second case we proceed with pattern matching on mn. Now, since $mn\ :\ \mathsf{suc}\ m \leqslant n$ the only possible case is leqSuc and we end up with

$$\begin{array}{llllll} trans\ \mathsf{zero} & m & n & (\mathsf{leqZero}\ m) & mn\ = & \mathsf{leqZero}\ n \\ trans\ (\mathsf{suc}\ k)\ (\mathsf{suc}\ m) & (\mathsf{suc}\ n) & (\mathsf{leqSuc}\ k\ m\ km) & (\mathsf{leqSuc}\ m\ n\ mn) & = ? & \end{array}$$

The remaining case is closed by an appeal to leqSuc and a recursive call.

$$\begin{array}{llllll} trans\ \mathsf{zero} & m & n & (\mathsf{leqZero}\ m) & mn\ = & \mathsf{leqZero}\ n \\ trans\ (\mathsf{suc}\ k)\ (\mathsf{suc}\ m) & (\mathsf{suc}\ n) & (\mathsf{leqSuc}\ k\ m\ km) & (\mathsf{leqSuc}\ m\ n\ mn) & = & \\ \quad \mathsf{leqSuc}\ k\ n\ (trans\ k\ m\ n\ km\ mn) & & & & & \end{array}$$

There are a number of interesting things to note here. First of all, as mentioned previously, when pattern matching on elements of the $\leqslant$ family the indices are instantiated. In this case, the patterns for the natural number arguments were refined even though we never explicitly pattern matched on them. This has the effect that the patterns become (seemingly) non-linear. In the last case above there are multiple occurrences of the variables k, m, and n. It is important to point out, however, that the repeated variables are exactly those that are necessary to make the left hand side well-typed.

The final thing to note is that we have a more refined notion of impossible patterns than you have for simple datatypes. Above we concluded that the constructor leqZero could not be used to build an element of $\mathsf{suc}\ m \leqslant n$. This is explained in detail in Section 2.1.8.

For now let us turn our attention to the non-linearity of patterns. The important observation is that the non-linearity arises from the instantiation of indices. In general we might not only get non-linear pattern but arbitrary terms in patterns. Consider the datatype *Imf* representing the property of being in the image of a function $f : A \to B$ (assuming some $A, B : \mathsf{Set}$ and $f : A \to B$):

$$\begin{array}{l} \textbf{data}\ Imf\ :\ B\ \to\ \mathsf{Set}\ \textbf{where} \\ \quad \mathsf{imf}\ :\ (x\ :\ A)\ \to\ Imf\ (f\ x) \end{array}$$

We can define the right inverse of f for a $y : B$ by pattern matching on a proof that y is in the image of f:

$$\begin{aligned}&\mathit{invf} \;:\; (y \;:\; B) \;\to\; \mathit{Imf}\ y \;\to\; A\\ &\mathit{invf}\ (f\ x)\ (\mathsf{imf}\ x) \;=\; x\end{aligned}$$

Here, pattern matching on the element of $\mathit{Imf}\ y$ instantiates y to $f\ x$ which is not a pattern at all, and there is no hope at runtime to check that the first argument matches $f\ x$. To solve this problem we distinguish between *accessible patterns* arising from explicit pattern matching, and *inaccessible patterns* arising from index instantiation as introduced by Goguen et al. [GMM06]. We augment the syntax for patterns with inaccessible patterns $\lfloor \mathit{term} \rfloor$[2]:

$$\mathit{pat} \quad ::= \quad x \;\mid\; \mathsf{c}\ \mathit{pat}^* \;\mid\; \lfloor \mathit{term} \rfloor$$

Making the inaccessible patterns explicit in the examples above we get

$$\begin{aligned}&\mathit{trans} \;:\; (k\ m\ n \;:\; \mathit{Nat}) \;\to\; k \leqslant m \;\to\; m \leqslant n \;\to\; k \leqslant n\\ &\mathit{trans}\ \lfloor\mathsf{zero}\rfloor\ \ \lfloor m\rfloor\ \ n\ \ (\mathsf{leqZero}\ m)\ \ \mathit{mn} \;=\; \mathsf{leqZero}\ n\\ &\mathit{trans}\ \lfloor\mathsf{suc}\ k\rfloor\ \ \lfloor\mathsf{suc}\ m\rfloor\ \ \lfloor\mathsf{suc}\ n\rfloor\ \ (\mathsf{leqSuc}\ k\ \lfloor m\rfloor\ \mathit{km})\ (\mathsf{leqSuc}\ m\ n\ \mathit{mn}) \;=\;\\ &\quad \mathsf{leqSuc}\ k\ n\ (\mathit{trans}\ k\ m\ n\ \mathit{km}\ \mathit{mn})\end{aligned}$$

$$\begin{aligned}&\mathit{invf} \;:\; (y \;:\; B) \;\to\; \mathit{Imf}\ y \;\to\; A\\ &\mathit{invf}\ \lfloor f\ x\rfloor\ (\mathsf{imf}\ x) \;=\; x\end{aligned}$$

Now the accessible parts of a pattern forms a well-formed linear pattern built by constructor applications and variables and the inaccessible patterns reference only variables bound in the accessible part. When computing the pattern matching at runtime only the accessible patterns need to be considered, the inaccessible patterns are guaranteed to match simply by the fact that the program is well-typed. Hence, the same compilation techniques that work for pattern matching in simply typed languages can be applied to pattern matching over inductive families.

In this chapter we give a detailed description of an algorithm for checking the correctness of functions defined by pattern matching over inductive families. There are two possible approaches to doing this: the external approach, taken by Coquand [Coq92] where correctness is verified by an external checker, and the internal approach, taken by Goguen et al. [GMM06], where correctness is verified by translation into a core theory. We choose the external approach since it allows us the luxury of working in the metatheory

[2]The concrete syntax for the inaccessible pattern $\lfloor t \rfloor$ in Agda is `.t` (see Chapter 5 for more information).

rather than in the theory itself when describing the type checking algorithm which makes things a bit easier. Our work is based on Coquand's algorithm, but where he describes how to incrementally construct a well-typed program we give a detailed algorithm for program recognition.

2.1 Type checking pattern match equations

In this section we present the type checking algorithm for systems of pattern match equations. Contrary to previous work [Coq92, GMM06] we allow equations to overlap and prioritise the rules from top to bottom. Operationally, however, we translate the system of equations to a case tree [Aug85]. This means that all equations might not hold as definitional equalities. Consider, for instance, the definition

$$
\begin{array}{lllll}
\_\sqcup\_ & : & \mathit{Nat} & \to \mathit{Nat} & \to \mathit{Nat} \\
x & \sqcup & \mathsf{zero} & = x & \\
\mathsf{zero} & \sqcup & y & = y & \\
\mathsf{suc}\, x & \sqcup & \mathsf{suc}\, y & = \mathsf{suc}\,(x \sqcup y) &
\end{array}
$$

Here, there is no way we could get both the first two equations to hold definitionally.

The algorithm works by first type checking each equation individually, and then checking that all cases are covered by translating the system into one that can be represented by a case tree.

We use the following conventions: u, v, w stand for well-typed terms, e for a potentially ill-typed term, p, q are patterns, σ, δ, γ are context mappings (substitutions), and Greek capital letters (Γ, Δ, ..) are contexts (telescopes).

2.1.1 Context mappings

A context mapping $\sigma : \Delta \to \Gamma$ is a list of patterns with $\Delta \vdash \sigma : \Gamma$ which is linear in the variables of Δ. This means that each variable in Δ occurs exactly once in an accessible position in σ. There are no restrictions on the inaccessible occurrences of a variable, however. If $\Gamma \vdash v : A$, then we can substitute v and A by σ, obtaining $\Delta \vdash v\sigma : A\sigma$. The identity mapping $\mathit{id} : \Gamma \to \Gamma$ is the list of variables in Γ. The singleton context mapping $[x := p] : \Gamma|_{x:=p} \to \Gamma$ is the list of variables in Γ where x has been replaced by p.

The context $\Gamma|_{x:=p}$ is defined by

$$
\frac{\Delta \sim \Delta^p \Delta_p \qquad \Gamma\Delta^p \vdash p : A \qquad \Gamma\Delta^p, x : A \vdash \Delta_p}{(\Gamma, x : A, \Delta)|_{x:=p} = \Gamma\Delta^p(\Delta_p[x := p])}
$$

We write $\Gamma \sim \Delta$ when Δ is a dependency preserving permutation of Γ. In this case we split the context after x into the part needed to type p (Δ^p) and the part depending on x (Δ_p).

We can lift a context mapping $\sigma : \Gamma \to \Delta$ to act on an extended telescope. We define $\sigma\uparrow^{\Theta} : \Gamma(\Theta\sigma) \to \Delta\Theta$ as the context mapping obtained by extending σ with the variables in Θ.

Given two context mappings $\delta : \Gamma \to \Delta$ and $\sigma : \Theta \to \Gamma$ we can form the composition $\delta \circ \sigma : \Theta \to \Delta$ by substituting δ by σ: $\delta \circ \sigma = \delta\sigma$.

2.1.2 Overview of the algorithm

The type checking algorithm takes a sequence of patterns $\bar{p}$ given by the user (the left hand side of one equation) and checks them against a telescope Γ (the types of the arguments to the function). If successful it computes a context Δ and context mapping $\sigma : \Delta \to \Gamma$, where Δ is the context of the variables bound in the left hand side, and σ is the type checked version of $\bar{p}$.

This is done by successively refining configurations $\langle \bar{q}, \delta : \Delta \to \Gamma \rangle$, starting with $\langle \bar{p}, id : \Gamma \to \Gamma \rangle$. The invariant is that $\bar{q}$ is expected to have type Δ, in other words $\bar{q}$ are the user patterns corresponding to the variables in δ. In each step we pick a constructor pattern in $\bar{q}$ and instantiate the corresponding variable in δ with the constructor applied to fresh variables. The algorithm terminates when $\bar{q}$ consists entirely of variables.

As seen in the beginning of the chapter, instantiating a variable with a constructor pattern involves unifying the datatype indices of the variable with those of the constructor pattern.

We continue by defining the three components of the type checking algorithm: matching, unification, and context splitting.

2.1.3 Matching

First we define how to match a sequence of patterns against a context mapping—remember that context mappings are simply lists of patterns. We write $\textsc{Match}(\sigma, \bar{p}) \implies \bar{q}$ for the successful matching of $\bar{p}$ against σ. If $\Theta \vdash \bar{p} : \Gamma$ and $\sigma : \Delta \to \Gamma$ then $\bar{q} : \Theta \to \Delta$, that is $\bar{q}$ instantiates the variables in σ with patterns from $\bar{p}$. Matching fails by throwing an exception which we write $\textsc{Match}(\sigma, \bar{p}) \Uparrow$. We write $\textsc{Match}(\sigma, \bar{p}) \overset{?}{\Longrightarrow}$ for a stuck matching, i.e. when neither $\textsc{Match}(\sigma, \bar{p}) \implies \bar{q}$ nor $\textsc{Match}(\sigma, \bar{p}) \Uparrow$. The rules are as

follows:

$$\frac{}{\textsc{Match}(x, p) \Longrightarrow [x := p]} \qquad \frac{}{\textsc{Match}(\lfloor u \rfloor, p) \Longrightarrow \varepsilon}$$

$$\frac{\textsc{Match}(\sigma, \bar{p}) \Longrightarrow \bar{q}}{\textsc{Match}(\mathsf{c}\,\sigma, \mathsf{c}\,\bar{p}) \Longrightarrow \bar{q}} \qquad \frac{\mathsf{c}_1 \neq \mathsf{c}_2}{\textsc{Match}(\mathsf{c}_1\,\sigma, \mathsf{c}_2\,\bar{p}) \Uparrow} \qquad \frac{}{\textsc{Match}(\varepsilon, \varepsilon) \Longrightarrow \varepsilon}$$

$$\frac{\textsc{Match}(p_1, p_2) \Longrightarrow \bar{q_1} \qquad \textsc{Match}(\sigma, \bar{p}) \Longrightarrow \bar{q_2}}{\textsc{Match}(p_1; \sigma, p_2; \bar{p}) \Longrightarrow \bar{q_1}; \bar{q_2}}$$

Note that anything matches an inaccessible pattern. This is reasonable since inaccessible patterns are guaranteed to match by the type system.

2.1.4 Unification

Unification is performed relative to a set of flexible variables, i.e. variables that are open for unification. In our case the flexible variables are those corresponding to inaccessible patterns in the input pattern, computed by $\textsc{Flexible}(\bar{p} : \Delta)$

$$\begin{array}{lcl} \textsc{Flexible}(\varepsilon : \varepsilon) & = & \emptyset \\ \textsc{Flexible}(\lfloor e \rfloor, \bar{p} : (x : A)\Delta) & = & \{x\} \cup \textsc{Flexible}(\bar{p} : \Delta) \\ \textsc{Flexible}(p; \bar{p} : (x : A)\Delta) & = & \textsc{Flexible}(\bar{p} : \Delta) \end{array}$$

The reason for keeping track of flexible variables is that we need to make sure that the context mapping generated by the algorithm corresponds to the patterns given by the user.

Upon successful unification a context mapping from a new context to the original context is produced. We write

$$\zeta, \Gamma \vdash \textsc{Unify}(u = v : A) \Longrightarrow \sigma : \Delta \to \Gamma$$

for the successful unification of u and v of type A in the context Γ with flexible variables ζ, resulting in the context mapping σ from the new context Δ to Γ. Intuitively Δ will be the context obtained by applying the unifier of u and v to Γ. As we shall see this might require reordering of Γ. A failed unification is written

$$\zeta, \Gamma \vdash \textsc{Unify}(u = v : A) \Uparrow$$

When faced with a problem which is too difficult unification will simply give up. We represent this by a stuck unification problem. For instance, the

unification of $x + y$ and $z + w$ with respect to x, y, z, and w will get stuck, since there is no unique solution. Note, however, that the problem can get unstuck if some of the variables are solved at a later stage.

We use the same notation for unifying a sequence of terms matching a telescope. The rules are presented in Figure 2.1. Three rules are of special interest to point out.

The rule (U-Conv) states that if u and v are convertible then they unify by the identity context mapping. This means that we can allow arbitrary terms in the indices as long as no unification is necessary.

The rule (U-Occ) causes unification to fail on cyclic equations such as $x = \mathsf{suc}\, x$. The set of accessible variables $\textsc{Acc}(p)$ in a pattern p is computed by

$$\begin{aligned} \textsc{Acc}(x) &= \{x\} \\ \textsc{Acc}(\mathsf{c}\,\bar{p}) &= \textsc{Acc}(\bar{p}) \\ \textsc{Acc}(\lfloor v \rfloor) &= \emptyset \end{aligned}$$

The rule containing most of the action is the (U-Var) rule. If x is a flexible variable and v is a term not containing x we can instantiate x to v.

2.1.5 Context splitting

The notion of context splitting was introduced by Coquand [Coq92] as a way to incrementally build a covering, i.e. a set of exhaustive patterns, for a context. A context $\Delta = \Delta_1(x : A)\Delta_2$ can be split along x if A is a datatype and we can figure out exactly which constructors can legally be used to build an element of A. This generates a set of new contexts where x has been instantiated with an application of each of the legal constructors to fresh variables.

If A is an ordinary inductive datatype all constructors are always legal, but in the case of inductive families it is a bit more interesting. Consider, for instance, the ordering relation on natural numbers given in the introduction to the chapter:

$$\begin{aligned} &\textbf{data}\ \_\leqslant\_ : \mathit{Nat} \to \mathit{Nat} \to \mathsf{Set}\ \textbf{where} \\ &\quad \mathsf{leqZero} : (n : \mathit{Nat}) \to \mathsf{zero} \leqslant n \\ &\quad \mathsf{leqSuc}\ \ : (n\ m : \mathit{Nat}) \to n \leqslant m \to \mathsf{suc}\, n \leqslant \mathsf{suc}\, m \end{aligned}$$

If $A = \mathsf{suc}\, n \leqslant \mathsf{suc}\, m$ then the $\mathsf{leqZero}$ constructor cannot be used to construct an element of A, so splitting only generates a single new context where x has been instantiated with an application of leqSuc. If, on the other hand,

$$\frac{x \in \zeta \qquad x \notin \mathrm{FV}(v)}{\zeta, \Gamma \vdash \textsc{Unify}(x = v : A) \Longrightarrow [x := \lfloor v \rfloor] : \Gamma|_{x:=v} \to \Gamma} \quad \text{(U-Var)}$$

$$\frac{\mathsf{c}_1 \neq \mathsf{c}_2}{\zeta, \Gamma \vdash \textsc{Unify}(\mathsf{c}_1\, \bar{u} = \mathsf{c}_2\, \bar{v} : A) \Uparrow} \quad \text{(U-Fail)}$$

$$\frac{x \in \textsc{Acc}(\bar{p})}{\zeta, \Gamma \vdash \textsc{Unify}(x = \mathsf{c}\, \bar{p} : A) \Uparrow} \quad \text{(U-Occ)}$$

$$\frac{\mathsf{c} : \Theta \to D\, \bar{w} \qquad \zeta, \Gamma \vdash \textsc{Unify}(\bar{u} = \bar{v} : \Theta) \Longrightarrow \sigma : \Gamma' \to \Gamma}{\zeta, \Gamma \vdash \textsc{Unify}(\mathsf{c}\, \bar{u} = \mathsf{c}\, \bar{v} : A) \Longrightarrow \sigma : \Gamma' \to \Gamma} \quad \text{(U-Con)}$$

$$\frac{}{\zeta, \Gamma \vdash \textsc{Unify}(\varepsilon = \varepsilon : \varepsilon) \Longrightarrow id : \Gamma \to \Gamma} \quad \text{(U-Empty)}$$

$$\frac{\begin{array}{c}\zeta, \Gamma \vdash \textsc{Unify}(u = v : A) \Longrightarrow \sigma_1 : \Gamma_1 \to \Gamma \\ \zeta, \Gamma_1 \vdash \textsc{Unify}(\bar{u}[\sigma_1] = \bar{v}[\sigma_1] : \Theta[x := u][\sigma_2]) \Longrightarrow \sigma_2 : \Gamma_2 \to \Gamma_1\end{array}}{\zeta, \Gamma \vdash \textsc{Unify}(u; \bar{u} = v; \bar{v} : (x : A)\Theta) \Longrightarrow \sigma_2\sigma_1 : \Gamma_2 \to \Gamma} \quad \text{(U-Tel)}$$

$$\frac{\Gamma \vdash u \simeq v \uparrow A}{\zeta, \Gamma \vdash \textsc{Unify}(u = v : A) \Longrightarrow id : \Gamma \to \Gamma} \quad \text{(U-Conv)}$$

Figure 2.1: Unification

$A = f\ n \leqslant m$ for some defined function f we cannot tell which constructors are legal and so splitting along x is not possible[3].

Splitting a context Δ along a variable x will, when successful, result in a family of context mappings $\sigma_j : \Gamma_j \to \Delta$ forming a covering of Δ. We will, however, define a more relaxed version of context splitting where we only check that a particular constructor can legally be used to instantiate a variable. Which variable to split along, and which constructor that should be used is determined by a sequence of user patterns. For a user pattern $\bar{p}$ supposedly of type Δ we write

$$\textsc{Split}(\bar{p},\ \Delta) \Longrightarrow \sigma : \Gamma \to \Delta$$

if there is a variable x in Δ corresponding to a constructor pattern in $\bar{p}$ that can be instantiated to that constructor. The result is a context mapping σ which performs the instantiation along with whatever further substitutions are necessary to make the whole thing well-typed. If we can find an illegal constructor in $\bar{p}$ we write

$$\textsc{Split}(\bar{p},\ \Delta) \Uparrow$$

The rules are given in Figure 2.2. Given a sequence of user patterns and a context we choose a constructor pattern $\mathsf{c}\ \bar{q}$ corresponding to a variable $x : A$ in the current context mapping. If A reduces to a datatype $D\ \bar{u}\ \bar{v}$, with $\bar{u}$ being the parameters of D and $\bar{v}$ being the indices, we check that c is indeed a constructor of D. If this is the case we lookup its type at the parameters $\bar{u}$ which gives us the type of the constructor arguments Θ and the indices $\bar{w}$ of its target. We unify $\bar{w}$ with $\bar{v}$ to obtain a context mapping $\delta : \Delta' \to \Delta_1\Theta$, for some context Δ'. The final context mapping is the composition δ with the instantiation of x to $\mathsf{c}\ \Theta\delta$, with liftings inserted at the right places.

The rule for failed splitting proceeds in the same way, but stops if unification fails. There is also a rule for failing when the constructor does not have the right type which we omit.

Using this relaxed context splitting operation we can define the standard splitting operation as introduced by Coquand which computes a covering set of context mappings over a context as follows:

[3]In some cases it might be possible to tell, but rather than resorting to complicated heuristics we chose the simpler approach of refusing to split.

$$\frac{\begin{array}{c} A \rightarrow_{whnf} D\ \bar{u}\ \bar{v} \\ D\ \bar{u} : \Xi \rightarrow \mathsf{Set} \qquad \mathsf{c}_{\bar{u}} : \Theta \rightarrow D\ \bar{u}\ \bar{w} \qquad \zeta = \textsc{Flexible}(\bar{p_1}; \bar{q} : \Delta_1\Theta) \\ \zeta, \Delta_1\Theta \vdash \textsc{Unify}(\bar{v} = \bar{w} : \Xi) \Longrightarrow \delta : \Delta' \rightarrow \Delta_1\Theta \\ \delta' = \delta\!\uparrow^{(x:A)} \circ [x := \mathsf{c}\ \Theta\delta] : \Delta' \rightarrow \Delta_1(x : A) \end{array}}{\textsc{Split}(\bar{p_1}; \mathsf{c}\ \bar{q}; \bar{p_2},\ \Delta_1(x : A)\Delta_2) \Longrightarrow \delta'\!\uparrow^{\Delta_2} : \Delta'(\Delta_2\delta') \rightarrow \Delta_1(x : A)\Delta_2}$$

$$\frac{\begin{array}{c} A \rightarrow_{whnf} D\ \bar{u}\ \bar{v} \qquad D\ \bar{u} : \Xi \rightarrow \mathsf{Set} \qquad \mathsf{c}_{\bar{u}} : \Theta \rightarrow D\ \bar{u}\ \bar{w} \\ \zeta = \textsc{Flexible}(\bar{p_1}; \bar{q} : \Delta_1\Theta) \qquad \zeta, \Delta_1\Theta \vdash \textsc{Unify}(\bar{v} = \bar{w} : \Xi) \Uparrow \end{array}}{\textsc{Split}(\bar{p_1}; \mathsf{c}\ \bar{q}; \bar{p_2},\ \Delta_1(x : A)\Delta_2) \Uparrow}$$

Figure 2.2: Configuration refinement rules

$$\frac{\begin{array}{l} A \rightarrow_{whnf} D\ \bar{u}\ \bar{v} \\ \forall \mathsf{c}_j \in \textsc{Constrs}(D). \\ \quad \mathsf{c}_{j\,\bar{u}} : \Theta_j \rightarrow D\ \bar{u}\ \bar{w} \\ \quad \bar{p_j} = \lfloor \Gamma_1 \rfloor ; \mathsf{c}\ \lfloor \Theta \rfloor ; \lfloor \Gamma_2 \rfloor \\ \quad \Phi_j = \begin{cases} \{\sigma_j\} & \text{if } \textsc{Split}(\bar{p_j},\ \Gamma_1(x : A)\Gamma_2) \Longrightarrow \sigma_j : \Gamma_j \rightarrow \Gamma \\ \emptyset & \text{if } \textsc{Split}(\bar{p_j},\ \Gamma_1(x : A)\Gamma_2) \Uparrow \end{cases} \end{array}}{\textsc{Split}_x(\Gamma_1(x : A)\Gamma_2) \Longrightarrow \bigcup_j \Phi_j}$$

If the context can be split along x then splitting returns the set of context mappings obtained by splitting with respect to each constructor in the datatype at x. We will use this splitting in Section 2.2 when we discuss the reduction behaviour of functions defined by pattern matching.

2.1.6 Type checking algorithm

As described in Section 2.1.2, the type checking algorithm builds a well-typed context mapping corresponding to the given user patterns by successively refining configurations in the form $\langle \bar{p}, \sigma : \Delta \rightarrow \Gamma \rangle$, where Γ is the type of the arguments to the function being checked, σ is the context mapping built so far, and $\bar{p}$ are the user patterns corresponding to the variables in σ. We write

$$\langle \bar{p}, \sigma : \Delta \rightarrow \Gamma \rangle \Longrightarrow \langle \bar{q}, \delta : \Theta \rightarrow \Gamma \rangle$$

for such a refinement. A configuration is refined by splitting the context as follows:

$$\frac{\textsc{Split}(\bar{p},\ \Delta) \Longrightarrow \delta : \Delta' \to \Delta \qquad \textsc{Match}(\delta, \bar{p}) \Longrightarrow \bar{p}'}{\langle \bar{p}, \sigma : \Delta \to \Gamma \rangle \Longrightarrow \langle \bar{p}', \sigma \circ \delta : \Delta' \to \Gamma \rangle}$$

New user patterns are computed by matching the old user patterns against the context mapping produced by splitting. This will extract the user patterns corresponding to the variables in Δ'. To wrap up, we define

$$\textsc{CheckPats}(\bar{p} : \Gamma) \Longrightarrow \sigma : \Delta \to \Gamma$$

to apply refinement repeatedly to the configuration $\langle \bar{p}, id : \Gamma \to \Gamma \rangle$ until only variables are left in the user pattern.

$$\frac{\langle \bar{p}, id : \Gamma \to \Gamma \rangle \Longrightarrow^* \langle \bar{x}, \sigma : \Delta \to \Gamma \rangle}{\textsc{CheckPats}(\bar{p} : \Gamma) \Longrightarrow \sigma : \Delta \to \Gamma}$$

Here $\Longrightarrow^*$ is the reflexive transitive closure of $\Longrightarrow$. Note that finding the right sequence of context splittings may require search. See Section 2.2 for an example.

2.1.7 Checking inaccessible patterns

When checking a left hand side the inaccessible parts of the given patterns are ignored. Instead we perform this check after the accessible part has been deemed correct. The reason for this is that the inaccessible patterns should be type correct in the context bound by the accessible patterns, and we do not know what this context is until we have checked the accessible part. So, given that $\textsc{CheckPats}(\bar{p} : \Gamma) \Longrightarrow \sigma : \Delta \to \Gamma$ we check the judgement

$\Delta \vdash \textsc{CheckInaccessible}(\bar{p} = \sigma : \Gamma)$ defined by

$$\frac{\Delta \vdash e \uparrow A \rightsquigarrow u \qquad \Delta \vdash u \simeq v \uparrow A}{\Delta \vdash \textsc{CheckInaccessible}(\lfloor e \rfloor = \lfloor v \rfloor : A)}$$

$$\frac{}{\Delta \vdash \textsc{CheckInaccessible}(x = x : A)}$$

$$\frac{\mathsf{c} : \Theta \to D\,\bar{w} \qquad \Delta \vdash \textsc{CheckInaccessible}(\bar{p} = \bar{q} : \Theta)}{\Delta \vdash \textsc{CheckInaccessible}(\mathsf{c}\,\bar{p} = \mathsf{c}\,\bar{q} : A)}$$

$$\frac{\begin{array}{c}\Delta \vdash \textsc{CheckInaccessible}(p = q : A) \\ \Delta \vdash \textsc{CheckInaccessible}(\bar{p} = \bar{q} : \Delta[x := q])\end{array}}{\Delta \vdash \textsc{CheckInaccessible}(p; \bar{p} = q; \bar{q} : (x : A)\Delta)}$$

$$\frac{}{\Delta \vdash \textsc{CheckInaccessible}(\varepsilon = \varepsilon : \varepsilon)}$$

Note that since we have checked the accessible part of the patterns we know that $\bar{p}$ and σ agrees on constructors and variable names.

This is all we need to check a left hand side. We define

$$\frac{\begin{array}{c}\textsc{CheckPats}(\bar{p} : \Gamma) \Longrightarrow \sigma : \Delta \to \Gamma \\ \Delta \vdash \textsc{CheckInaccessible}(\bar{p} = \sigma : \Gamma)\end{array}}{\textsc{CheckLhs}(\bar{p} : \Gamma) \Longrightarrow \sigma : \Delta \to \Gamma}$$

2.1.8 Refuting elements of empty types

In many previous presentations [Coq92, McB99, SP03] coverage checking is undecidable. This is due to the fact that splitting on a caseless datatype does not leave any evidence in the program—it simply makes the whole branch disappear. To solve this problem we follow the same approach taken by Goguen et al. [GMM06] and require programs to contain explicit dismissal of elements in empty types.

First we make a distinction between empty types and caseless types. Informally we say that an empty type is a type with no closed inhabitants, whereas a caseless datatype is a type with no constructor headed *open* inhabitants. For instance, $\bot$ is caseless, while $\bot'$ is not:

```
data ⊥  : Set where
data ⊥′ : Set where
```

$$\mathsf{bot} \;:\; \bot \;\to\; \bot'$$

Another example of an empty but not caseless datatype is *Inf*:

$$\begin{array}{l}\textbf{data}\ \mathit{Inf}\ :\ \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{inf}\ :\ \mathit{Inf}\ \to\ \mathit{Inf}\end{array}$$

In the presence of inductive families the set of caseless datatypes are more interesting. For instance, the type $\mathsf{suc}\ n \leqslant \mathsf{zero}$ is caseless for the definition of $\leqslant$ given in the introduction to this chapter.

$$\begin{array}{l}\textbf{data}\ \_\leqslant\_\ :\ \mathit{Nat}\ \to\ \mathit{Nat}\ \to\ \mathsf{Set}\ \textbf{where}\\ \quad \mathsf{leqZero}\ :\ (n\ :\ \mathit{Nat})\ \to\ \mathsf{zero} \leqslant n\\ \quad \mathsf{leqSuc}\ \ :\ (n\ m\ :\ \mathit{Nat})\ \to\ n \leqslant m\ \to\ \mathsf{suc}\ n \leqslant \mathsf{suc}\ m\end{array}$$

We only consider caselessness for (types convertible with) datatypes. To see why consider the set-valued function F defined by

$$\begin{array}{l}F\ :\ \mathit{Nat}\ \to\ \mathsf{Set}\\ F\ \mathsf{zero}\ \ \ \ =\ \bot\\ F\ (\mathsf{suc}\ n) =\ F\ n\end{array}$$

According to our informal definition $F\ n$ is a caseless type, however, it is unreasonable to expect a type checker to be able to see this—indeed it is undecidable in general.

To check that a datatype is caseless we can use the context splitting facilities already developed. We define $\Gamma \vdash \textsc{Caseless}(A)$ by

$$\frac{\textsc{Split}_x(\Gamma(x : A)) \Longrightarrow \emptyset}{\Gamma \vdash \textsc{Caseless}(A)}$$

That is, A is caseless in the context Γ if splitting $\Gamma(x : A)$ along x results in an empty covering.

2.1.9 Checking the right hand side

The syntax of right hand sides is

$$\mathit{rhs} \;\;::=\;\; = \mathit{term} \mid \Cap\ \bar{x}$$

If a left hand side binds variables $\bar{x}$ of caseless types the right hand side $\Cap\ \bar{x}$ refutes $\bar{x}$. Note that it is always enough to refute a single variable, but for documentation purposes it might be nice to be able to refute more than one.

We can now give the rules for checking a clause in a function definition. We write $\textsc{CheckClause}(f\,\bar{p}\,rhs : \Gamma \to A)$ for the checking of $f\,\bar{p}\,rhs$ against the type $\Gamma \to A$.

$$\frac{\textsc{CheckLhs}(\bar{p} : \Gamma) \Longrightarrow \sigma : \Delta \to \Gamma \qquad \Delta \vdash e \uparrow A\sigma \rightsquigarrow v}{\textsc{CheckClause}(f\,\bar{p} = e : \Gamma \to A)}$$

$$\frac{\textsc{CheckLhs}(\bar{p} : \Gamma) \Longrightarrow \sigma : \Delta \to \Gamma \qquad \forall i.\ \Delta \vdash \textsc{Caseless}(\Delta(x_i))}{\textsc{CheckClause}(f\,\bar{p} \Cap \bar{x} : \Gamma \to A)}$$

To save the user from inventing names for refuted variables we extend the syntax of patterns with a special pattern $\emptyset$, the meaning of which is an anonymous variable that is implicitly refuted in the right hand side[4]. For instance,

$$\begin{aligned} &f\ :\ (n\ :\ \mathit{Nat})\ \to\ \mathsf{suc}\ n\ \leqslant\ \mathsf{zero}\ \to\ (A\ :\ \mathsf{Set})\ \to\ A \\ &f\ n\ \emptyset\ A \end{aligned}$$

is the user syntax for

$$\begin{aligned} &f\ :\ (n\ :\ \mathit{Nat})\ \to\ \mathsf{suc}\ n\ \leqslant\ \mathsf{zero}\ \to\ (A\ :\ \mathsf{Set})\ \to\ A \\ &f\ n\ x\ A\ \Cap\ x \end{aligned}$$

2.2 Coverage checking

In previous work [Coq92, GMM06] definitions have been restricted to non-overlapping patterns corresponding to a covering of the argument context. In this work we have relaxed this requirement and, so far, only required that the clauses of a definition can be obtained by our relaxed form of context splitting. This means that we allow overlapping clauses. For instance,

$$\begin{aligned} &\_\sqcup\_\ :\ \mathit{Nat}\ \to\ \mathit{Nat}\ \to\ \mathit{Nat} \\ &x\ \ \sqcup\ \mathsf{zero}\ =\ x \\ &\mathsf{zero}\ \sqcup\ y\ \ =\ y \\ &\mathsf{suc}\ x\ \sqcup\ \mathsf{suc}\ y\ =\ \mathsf{suc}\ (x\ \sqcup\ y) \end{aligned}$$

where the first two clauses overlap, or

$$\_==\_\ :\ \mathit{Nat}\ \to\ \mathit{Nat}\ \to\ \mathit{Bool}$$

[4]The concrete syntax for $\emptyset$ in Agda is `()`.

$$
\begin{array}{lllll}
\mathsf{zero} & == & \mathsf{zero} & = & \mathsf{true} \\
\mathsf{suc}\ x & == & \mathsf{suc}\ y & = & x\ ==\ y \\
x & == & y & = & \mathsf{false}
\end{array}
$$

where we have a catch-all case at the end. Allowing this kind of overlap can reduce the number of clauses a lot—in the case of the equality function, from quadratic in the number of constructors to linear. The drawback of allowing overlapping patterns is that the order of the clauses is significant—when two clauses overlap, the top-most clause will take priority. In the presence of overlapping patterns it is clear that we cannot expect the clauses of a function to hold as definitional equalities. For instance, it is obviously not the case that $x == y = \mathsf{false}$ for arbitrary x and y. Perhaps more surprisingly, the same holds even when clauses are disjoint. The notorious example is the majority function due to Gérard Berry defined as

$$
\begin{array}{lllll}
maj & : & Bool \to Bool \to Bool & \to & Bool \\
maj & \mathsf{true} & \mathsf{true} & \mathsf{true} & = \mathsf{true} \\
maj & \mathsf{true} & \mathsf{false} & z & = z \\
maj & \mathsf{false} & y & \mathsf{true} & = y \\
maj & x & \mathsf{true} & \mathsf{false} & = x \\
maj & \mathsf{false} & \mathsf{false} & \mathsf{false} & = \mathsf{false}
\end{array}
$$

The clauses are clearly disjoint and exhaustive, yet there is no covering corresponding to this definition. In other words, there is no way we could get all five equations as definitional equalities in our core type theory.

In order to guarantee conservativity with respect to the core theory we need to show how to compute a single covering from a sequence of exhaustive but possibly overlapping clauses. This also guarantees that we can compile the pattern matching to an efficient case tree [Aug85]. The idea is to perform the same context splittings as the individual clauses, giving priority to earlier clauses over later clauses. In order to keep the algorithm predictable we make it incomplete. Consider the following contrived example:

$$
\begin{array}{l}
dbl\ :\ Nat\ \to\ Nat \\
dbl\ \mathsf{zero}\ \ \ = \mathsf{zero} \\
dbl\ (\mathsf{suc}\ n) = \mathsf{suc}\ (\mathsf{suc}\ (dbl\ n)) \\
\textbf{data}\ \_\overset{?}{=}\_\ (n\ :\ Nat)\ :\ Nat\ \to\ \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{eq}\ \ :\ n \overset{?}{=} n \\
\quad \mathsf{neq}\ :\ (m\ :\ Nat)\ \to\ n \overset{?}{=} m \\
\textbf{data}\ Even\ :\ Nat\ \to\ \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{even}\ :\ (m\ :\ Nat)\ \to\ D\ (dbl\ m) \\
\\
f\ :\ (n\ m\ :\ Nat)(x\ :\ Even\ m)(p\ :\ m \overset{?}{=} \mathsf{suc}\ n)\ \to\ Nat
\end{array}
$$

$$
\begin{array}{llll}
f\ n & m & x & (\mathsf{neq}\ \lfloor \mathsf{suc}\ m \rfloor) = \ldots \\
f\ \lfloor \mathsf{suc}\ (\mathit{dbl}\ m) \rfloor & \lfloor \mathsf{suc}\ (\mathsf{suc}\ (\mathit{dbl}\ m)) \rfloor & (\mathsf{even}\ (\mathsf{suc}\ m))\ \mathsf{eq} & = \ldots
\end{array}
$$

In order to obtain the second clause it is necessary to first split on x, but since the first clause only splits on p that is what our algorithm will start with. In the eq case we will then have the context $(n\ :\ \mathit{Nat})(x\ :\ \mathit{Even}\ (\mathsf{suc}\ n))$ where we would like to split on x. This is not possible since unification gives up on $\mathit{dbl}\ m\ =\ \mathsf{suc}\ n$.

Rather than report an error in this case, which is what we do, one could imagine backtracking and trying to split in a different order. The drawback with this approach is that it will be very hard for the user to predict what the resulting covering will be. With our approach this is much easier. Another option is of course to give up on overlapping patterns and use the algorithm outlined by Coquand [Coq92], but as we have seen overlapping cases can be quite handy at times.

Another observation is that with this algorithm it is not possible to recreate all splittings. Consider the following version of the majority function:

$$
\begin{array}{llll}
\mathit{maj}\ x & \mathsf{false}\ \mathsf{false} & = & \mathsf{false} \\
\mathit{maj}\ x & \mathsf{true}\ \mathsf{false} & = & x \\
\mathit{maj}\ \mathsf{false}\ x & \mathsf{true} & = & x \\
\mathit{maj}\ \mathsf{true}\ x & \mathsf{true} & = & \mathsf{true}
\end{array}
$$

This version corresponds exactly to the covering obtained by first splitting on the third argument and then in the false case splitting on the second argument and in the true case on the first argument. There is, however, no way of reordering the clauses to have our algorithm start by splitting on the third argument. On the other hand, it is easy to get this behaviour by introducing two helper functions, so we have not lost any expressivity.

2.2.1 Coverage algorithm

If $\textsc{Match}(\bar{p}, \bar{v}) \stackrel{?}{\Longrightarrow}$ then there is a non-empty sequence of neutral terms in $\bar{v}$ which are being matched against constructor patterns, and hence cause the matching to get stuck. We denote these terms by $\textsc{Blockers}(\bar{p}, \bar{v})$.

Now we define a *clause* C for a function $f : \Gamma \to A$ to be a context Δ, a context mapping $\sigma : \Delta \to \Gamma$, and a right hand side $\Delta \vdash \mathit{rhs} : A\sigma$. We leave the context Δ implicit, since it can be deduced from σ. Given the list of clauses provided by the user (which have been deemed proper clauses by the type checker) we compute a new set of clauses corresponding to a covering of the argument context. We write $\textsc{Covering}(\bar{C}, \delta : \Delta \to \Gamma) \Longrightarrow \bar{C}'$ where

$\bar{C}$ are the user clauses, δ is the current neighbourhood (δ starts out as $id : \Gamma \to \Gamma$), and $\bar{C}'$ are the computed clauses. The rules are

$$\frac{C_i = \langle \sigma_i, rhs_i \rangle \qquad \textsc{Match}(\sigma_i, \delta) \Longrightarrow \rho \quad \forall j < i.\ C_j = \langle \sigma_j, rhs_j \rangle \ \wedge\ \textsc{Match}(\sigma_j, \delta) \Uparrow}{\textsc{Covering}(\bar{C}, \delta : \Delta \to \Gamma) \Longrightarrow \langle \delta, rhs_i[\rho] \rangle} \quad (\textsc{Match})$$

$$\frac{\forall\, i.\ C_i = \langle \sigma_i, rhs_i \rangle \wedge \textsc{Match}(\sigma_i, \delta) \Uparrow}{\textsc{Covering}(\bar{C}, \delta : \Delta \to \Gamma) \Uparrow} \quad (\textsc{Missed})$$

$$\frac{\begin{array}{c}\langle \sigma, rhs \rangle \in \bar{C} \qquad \textsc{Match}(\sigma, \delta) \stackrel{?}{\Longrightarrow} \\ x \in \textsc{Blockers}(\sigma, \delta) \qquad \textsc{Split}_x(\Delta) \Longrightarrow \{\delta_j : \Delta_j \to \Delta \mid j\} \\ \forall j.\ \textsc{Covering}(\bar{C}, \delta\delta_j : \Delta_j \to \Gamma) \Longrightarrow \bar{C}_j \end{array}}{\textsc{Covering}(\bar{C}, \delta : \Delta \to \Gamma) \Longrightarrow \bigcup_j \bar{C}_j} \quad (\textsc{Split})$$

Basically the algorithm works by splitting the context until the current neighbourhood matches one of the original clauses (Match). In order to get the first match semantics we require that all earlier clauses result in a match failure. If the current neighbourhood fails to match all the given clauses they are not exhaustive and we terminate with coverage failure (Missed). If matching is inconclusive we split along one of the blocking variables and proceed recursively with the resulting neighbourhoods (Split). To improve the readability of the rule we do not specify how to pick σ and x, but in practice we pick the first σ and x which admit a split.

Let us look at the algorithm in action for the $\sqcup$ function defined above. The clauses are

$$\begin{array}{lll} \langle\ x \sqcup \mathsf{zero}, & x & \rangle \\ \langle\ \mathsf{zero} \sqcup y, & y & \rangle \\ \langle\ \mathsf{suc}\ x \sqcup \mathsf{suc}\ y, & \mathsf{suc}\ (x \sqcup y) & \rangle \end{array}$$

and we start out with the neighbourhood $x;\ y : (x\ y : Nat) \to (x\ y : Nat)$. Since we do not match any clause we apply the (Split) rule. Matching gets stuck on all clauses and the only blocker of the first clause is y, so we split on y. This yields the two neighbourhoods

$$\begin{array}{lll} \delta_1 = x; \mathsf{zero} & : (x : Nat) \to (x\ y : Nat) \\ \delta_2 = x; (\mathsf{suc}\ y) & : (x\ y : Nat) \to (x\ y : Nat) \end{array}$$

In the first case we can apply the (Match) rule since δ_1 matches the first clause with the identity substitution. The resulting clause is $\langle x\ \mathsf{zero}, x \rangle$. In

the second case matching against the first clause fails, but matching against the other two clauses is inconclusive. Hence we apply the (Split) rule splitting along x obtaining the neighbourhoods (after composition with δ_2)

$$\begin{array}{lll} \delta_3 = \mathsf{zero}; & (\mathsf{suc}\ y) : (y : Nat) & \to (x\ y : Nat) \\ \delta_4 = (\mathsf{suc}\ x); & (\mathsf{suc}\ y) : (x\ y : Nat) & \to (x\ y : Nat) \end{array}$$

Now δ_3 matches the second clause with the substitution $[y := \mathsf{suc}\ y]$ and δ_4 matches the third clause with the identity substitution so we produce the clauses $\langle \mathsf{zero}\ (\mathsf{suc}\ y),\ \mathsf{suc}\ y \rangle$ and $\langle (\mathsf{suc}\ x)\ (\mathsf{suc}\ y),\ \mathsf{suc}\ y \rangle$. The result is the following covering:

$$\begin{array}{lll} x & \sqcup\ \mathsf{zero} & = x \\ \mathsf{zero} & \sqcup\ \mathsf{suc}\ y & = \mathsf{suc}\ y \\ \mathsf{suc}\ x & \sqcup\ \mathsf{suc}\ y & = \mathsf{suc}\ (x \sqcup y) \end{array}$$

2.2.2 Uniqueness of identity proofs

As mentioned in Section 1.5.2 the pattern matching presented in this section can be reduced to elimination rules provided we have uniqueness of identity proofs (the K axiom [HS94]). This was shown by McBride [McB99, MM04a, GMM06] and this is how pattern matching is treated in Epigram [McB07].

To see where the K axiom is used let us walk through the type checking of its definition by pattern matching. Recall

$$\begin{array}{l} \textbf{data}\ Id\ (A : \mathsf{Set})(x : A) : A \to \mathsf{Set}\ \textbf{where} \\ \quad \mathsf{refl} : Id\ A\ x\ x \end{array}$$

To simplify matters we assume $(A : \mathsf{Set})(x : A)(P : Id\ A\ x\ x \to \mathsf{Set})$ and define

$$\begin{array}{l} K : (pr : P\ \mathsf{refl})(p : Id\ A\ x\ x) \to P\ p \\ K\ pr\ \mathsf{refl} = pr \end{array}$$

Checking the left hand side of this definition will involve a single splitting of the context $(pr : P\ \mathsf{refl})(p : Id\ A\ x\ x)$ along p with the expected constructor refl. The derivation is

$$\frac{\begin{array}{c} Id\ A\ x : A \to \mathsf{Set} \qquad \mathsf{refl}_{A;x} : Id\ A\ x\ x \\ \emptyset, (pr : P\ \mathsf{refl}) \vdash \textsc{Unify}(x = x : A) \Longrightarrow id : (pr : P\ \mathsf{refl}) \to (pr : P\ \mathsf{refl}) \end{array}}{\begin{array}{c} \textsc{Split}(pr; \mathsf{refl},\ (pr : P\ \mathsf{refl})(p : Id\ A\ x\ x)) \Longrightarrow \\ [p := \mathsf{refl}] : (pr : P\ \mathsf{refl}) \to (pr : P\ \mathsf{refl})(p : Id\ A\ x\ x) \end{array}}$$

The only thing happening in this derivation is that we unify x with itself. This is exactly where K is needed when translating to elimination rules. In order to be allowed to discard trivial equations, such as $x = x$, it is necessary to know that the only possible proof is refl.

2.3 The with construct

The *with* construct, introduced by McBride and McKinna [MM04a], allows analysis of intermediate results to be performed on the left hand side of a function definition rather than on the right hand side. In the presence of inductive families this is a very powerful tool which among other things makes it possible to roll your own case analyses (see Section 2.3.1).

The syntax for *with* is similar to that of McBride and McKinna, their *unzip* example look as follows:

$$
\begin{aligned}
&\mathit{unzip} \;:\; \{A\ B \;:\; \mathsf{Set}\}\{n \;:\; \mathit{Nat}\} \;\rightarrow \\
&\qquad \mathit{Vec}\,(A \times B)\,n \;\rightarrow\; \mathit{Vec}\,A\,n \times \mathit{Vec}\,B\,n \\
&\mathit{unzip}\,\varepsilon \;=\; \langle\, \varepsilon,\, \varepsilon \,\rangle \\
&\mathit{unzip}\,(\langle\, x,\, y \,\rangle \;::\; \mathit{xys}) \;\mathbf{with}\; \mathit{unzip}\;\mathit{xys} \\
&\mathit{unzip}\,(\langle\, x,\, y \,\rangle \;::\; \mathit{xys}) \qquad \mid \langle\, \mathit{xs},\, \mathit{ys} \,\rangle \;=\; \langle\, x \;::\; \mathit{xs},\, y \;::\; \mathit{ys} \rangle
\end{aligned}
$$

A *with* in effect adds an extra argument to a function. This argument is treated just as any previous argument and after the addition pattern matching can proceed as normal. The extent of a *with*-clause is determined by counting the number of additional arguments.

Behind the scenes, each *with*-clause is translated to an auxiliary function. More precisely, given a definition

$$
\begin{aligned}
&f \;:\; \Gamma \;\rightarrow\; B \\
&f\ \bar{p}\ \mathbf{with}\ e \\
&f\ \bar{p}_1 \;\mid q_1 \;=\; e_1 \\
&\vdots \\
&f\ \bar{p}_n \;\mid q_n \;=\; e_n
\end{aligned}
$$

where $\bar{p} : \Delta \rightarrow \Gamma$ and $\Delta \vdash e : A$, we first partition the context Δ in Δ_1 and Δ_2 where Δ_1 is the smallest part of Δ necessary to type e. Next we abstract all syntactic occurrences of e from $\Delta_2 \rightarrow B$ obtaining a type $\Delta_2' \rightarrow B'$ with $\Delta_2 = \Delta_2'[x := e]$ and $B = B'[x := e]$. The patterns $\bar{p}_i$ must be an instance of $\bar{p}$ so we check that $\textsc{Match}(\bar{p}, \bar{p}_i) \Longrightarrow \bar{p}_i'$. The definition of the auxiliary function is

$$
f' \;:\; \Delta_1 \;\rightarrow\; (x \;:\; A) \;\rightarrow\; \Delta_2' \;\rightarrow\; B'
$$

$$
\begin{aligned}
&f'\ \bar{p}'_1\ q_1 \ = \ e_1\\
&\ldots\\
&f'\ \bar{p}'_n\ q_n \ = \ e_n
\end{aligned}
$$

and we check that this constitutes a valid definition. It is important to check that the abstracted type is well-formed, since this is not necessarily the case. For instance, abstracting over the first projection of a dependent pair might not be well-typed without also abstracting over the second projection, since the first projection occurs in the type of the second projection. To abstract more than one expression at once they are separated by bars, like so:

$$
\begin{aligned}
&\mathit{many} \ : \ (x \ : \ (n \ : \ \mathit{Nat}) \ \times \ (n \ \leqslant \ \mathsf{zero})) \ \rightarrow \ \mathit{Nat}\\
&\mathit{many}\ x\ \mathbf{with}\ \pi_1\ x \ \mid \ \pi_2\ x\\
&\mathit{many}\ x \mid \ \lfloor \mathsf{zero} \rfloor \ \mid \mathsf{leqZero} \ = \ \mathsf{zero}
\end{aligned}
$$

2.3.1 Examples

Filtering lists

Abstracting syntactic occurrences of the analysed expression comes in very handy when reasoning about functions defined by *with*. Consider the *filter* function which removes all elements not satisfying a given predicate from a list.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{List}\ (A \ : \ \mathsf{Set}) \ : \ \mathsf{Set}\ \mathbf{where}\\
&\quad \varepsilon \quad : \ \mathit{List}\ A\\
&\quad \_::\_ \ : \ A \ \rightarrow \ \mathit{List}\ A \ \rightarrow \ \mathit{List}\ A\\
&\mathit{filter} \ : \ \{A \ : \ \mathsf{Set}\} \ \rightarrow \ (A \ \rightarrow \ \mathit{Bool}) \ \rightarrow \ \mathit{List}\ A \ \rightarrow \ \mathit{List}\ A\\
&\mathit{filter}\ p\ \varepsilon \ = \ \varepsilon\\
&\mathit{filter}\ p\ (x \ :: \ \mathit{xs})\ \mathbf{with}\ p\ x\\
&\mathit{filter}\ p\ (x \ :: \ \mathit{xs}) \ \mid \ \mathsf{true} \ = \ x \ :: \ \mathit{filter}\ p\ \mathit{xs}\\
&\mathit{filter}\ p\ (x \ :: \ \mathit{xs}) \ \mid \ \mathsf{false} = \ \mathit{filter}\ p\ \mathit{xs}
\end{aligned}
$$

Suppose we want to prove that the filtered list is a sublist of the original list, i.e. that all elements of the filtered list appears in the original list in the same order. We might define

$$
\begin{aligned}
&\mathbf{data}\ \_\subseteq\_\ \{A \ : \ \mathsf{Set}\} \ : \ \mathit{List}\ A \ \rightarrow \ \mathit{List}\ A \ \rightarrow \ \mathsf{Set}\ \mathbf{where}\\
&\quad \mathsf{stop} \ : \ \varepsilon \ \subseteq \ \varepsilon\\
&\quad \mathsf{keep} \ : \ \{x \ : \ A\}\{\mathit{xs}\ \mathit{ys} \ : \ \mathit{List}\ A\} \ \rightarrow \ \mathit{xs} \ \subseteq \ \mathit{ys} \ \rightarrow \ (x \ :: \ \mathit{xs}) \ \subseteq \ (x \ :: \ \mathit{ys})\\
&\quad \mathsf{skip} \ : \ \{y \ : \ A\}\{\mathit{xs}\ \mathit{ys} \ : \ \mathit{List}\ A\} \ \rightarrow \ \mathit{xs} \ \subseteq \ \mathit{ys} \ \rightarrow \ \mathit{xs} \ \subseteq \ (y \ :: \ \mathit{ys})\\
&\mathit{sublist} \ : \ \{A \ : \ \mathit{Set}\}(p \ : \ A \ \rightarrow \ \mathit{Bool})(\mathit{xs} \ : \ \mathit{List}\ A) \ \rightarrow \ \mathit{filter}\ p\ \mathit{xs} \ \subseteq \ \mathit{xs}\\
&\mathit{sublist}\ p\ \varepsilon \ = \ \mathsf{stop}
\end{aligned}
$$

```
sublist p (x :: xs) with p x
sublist p (x :: xs) | true  = keep (sublist p xs)
sublist p (x :: xs) | false = skip (sublist p xs)
```

To see that this works let us look at the auxiliary functions generated for our two with-clauses. For the *filter* function we get

```
filter′ : {A : Set} →
          (A → Bool) → A → List A → Bool → List A
filter′ p x xs true  = x :: filter p xs
filter′ p x xs false = filter p xs
```

Nothing very exciting happens here, but in the proof things get more interesting. In the ::-case the goal type is *filter* $p\ (x :: xs) \subseteq (x :: xs)$ which reduces to *filter′* $p\ x\ xs\ (p\ x) \subseteq (x :: xs)$, so when constructing the type of the auxiliary function there is an occurrence of $p\ x$ to abstract over. The generated function is

```
sublist′ : {A : Set}(p : A → Bool)(x : A)(xs : List A)
           (b : Bool) → filter′ p x xs b ⊆ (x :: xs)
sublist′ p x xs true  = keep (sublist p xs)
sublist′ p x xs false = skip (sublist p xs)
```

Since we abstracted over $p\ x$ the call to *filter′* will reduce when we pattern match on b. This is what makes the proof go through.

Rewriting using *with*

The *with* construct can be used as a rewriting tool. Recall the identity type (here with the type A implicit):

```
data _==_ {A : Set}(x : A) : A → Set where
  refl : x == x
```

In general we cannot pattern match on an element $eq : lhs == rhs$, since the unification algorithm might not be able to unify *lhs* and *rhs*. But, if we abstract over one of the sides, turning it into a fresh variable, unification will have no problems. For instance, to prove associativity of addition (defined by recursion over the first argument) we can write:

```
assoc : (x y z : Nat) → x + (y + z) == (x + y) + z
assoc zero    y z = refl
assoc (suc x) y z with x + (y + z) | assoc x y z
assoc (suc x) y z | ⌊(x + y) + z⌋  | refl = refl
```

The parity of a natural number

We mentioned above that the with construct can be used to emulate non-standard pattern matching. Here is an example which lets you match on a natural number being either $2 * k$ or $2 * k + 1$ for some k.

We first define a view datatype *Parity* with one constructor for each of our two cases.

```
data Parity : Nat → Set where
  even : (k : Nat) → Parity (2 * k)
  odd  : (k : Nat) → Parity (2 * k + 1)
```

The next step is to show that any number supports the *Parity* view. Note how we use the view in the recursive case.

```
parity : (n : Nat) → Parity n
parity zero = even zero
parity (suc n) with parity n
parity (suc ⌊2 * k⌋)     | even k = odd k
parity (suc ⌊2 * k + 1⌋) | odd k  = even (k + 1)
```

Now we can, for instance, define the function *half* very elegantly.

```
half : Nat → Nat
half n with parity n
half ⌊2 * k⌋     | even k = k
half ⌊2 * k + 1⌋ | odd k  = k
```

The concept of views in this form was introduced by McBride and McKinna [MM04a] and they take it one step further, allowing you to omit the patterns for the view datatype.

Chapter 3

Metavariables

In this chapter we present a type checking algorithm for a dependently typed logical framework extended with metavariables standing for as yet unknown terms. The chapter is based on an unpublished paper written together with Catarina Coquand [NC07].

It is common for frameworks supporting metavariables to accept that unification creates substitutions that are not well-typed [Dow01, Ell89, Pym90], but we give a novel approach to the treatment of metavariables where well-typedness of substitutions is guaranteed. To ensure type correctness the type checker creates well-typed approximations of the terms being type checked. We use a restricted form of pattern unification, but we believe that the results carry over to other unification algorithms. We prove that the algorithm is sound and terminating.

3.1 Introduction

Systems based on proposition-as-types provide an elegant approach to interactive proof assistants: the problem of proof checking is reduced to type checking and these systems combine in a natural way deduction and computation. A well understood formulation relies on lambda calculus with dependent types, [NPS90, Bar92b, dB91a]. The main problem is then checking the judgement $s : A$ expressing that a given term (proof), s, is of type (is a correct proof of the proposition) A.

A type checking algorithm can be naturally divided in two stages [dB91a]. In the first stage we go through the terms and whenever we type check a term s of type A against a type B we collect the equality constraint $A = B$. In the second phase we check these constraints by verifying that the terms are convertible with each other. With dependent types it is important to check

the constraints in the right order, and to fail as soon as an equality is invalid, since well typedness of a constraint may depend on previous constraints being satisfied.

For representing proof search in these frameworks it is convenient to extend the notion of terms with *metavariables* that stands for yet undetermined terms (proofs). Metavariables are also useful for structure editing, as placeholders for information to be filled in by the user. In this paper we will however focus on type reconstruction where metavariables are used for representing omitted information that can be recovered from typing constraints through unification.

When adding metavariables, equality checking gets more complicated, since we cannot always decide the validity of an equality, and we may be forced to keep it as a constraint. This is well-known in higher order unification [Hue75]: the constraint $? \; 0 = 0$ has two solutions $? = \lambda x.x$ and $? = \lambda x.0$. This appears also in type theory with constraints of the form $F \; ? = \mathit{Bool}$ where F is defined by computation rules. The fact that we type check modulo yet unsolved constraints can lead to ill-typed terms. For instance, consider the type-checking problem

$$\lambda g.\ g\ 0 : ((x : F\ ?) \to F\ (\neg\ x)) \to \mathit{Nat}$$

where

$$? \ : \mathit{Bool}$$
$$0 \ : \mathit{Nat}$$
$$F : \mathit{Bool} \to \mathsf{Set}$$
$$F\ \mathit{false} = \mathit{Nat}$$
$$F\ \mathit{true} \ = \mathit{Bool}$$

First we check that

$$((x : F\ ?) \to F\ (\neg\ x)) \to \mathit{Nat}$$

is a well-formed type, which generates the constraint $F \; ? = \mathit{Bool}$, since the term $\neg\ x$ forces x to be of type Bool. Checking

$$\lambda g.\ g\ 0 : ((x : F\ ?) \to F\ (\neg\ x)) \to \mathit{Nat}$$

then generates the constraints

$$F\ ? \quad\ = \mathit{Nat}$$
$$F\ (\neg\ 0) = \mathit{Nat}$$

which contains an ill-typed term.

This problem has some negative consequence for the type checking algorithm. With dependent types, verifying convertibility between two terms relies on normalising these terms, which is only safe if these terms are well-typed. But, as we have seen, in presence of metavariables, we may not be sure that these terms are well typed, and thus, the type checker may loop. It is still the case however that if all constraints can be solved we have a correct solution; so we have some form of partial correctness and this is indeed

the approach taken in Alf [MN94] and the previous version of Agda [CC99]. Elliot [Ell89] has a similar problem of generating ill-typed terms, but in his context this is less problematic, since these terms can still be shown to be normalisable in the logical framework he uses, which is more restricted than the one we consider. Another problem is that when we get a constraint of the form $? = s$, we cannot be sure that s is a solution for $?$, since we are not sure that s is well-typed. In previous work [MN94, CC99, Muñ01] this difficulty is avoided by re-type checking s at this point, which is costly.

Nanevski et al. describes a modal type theory [NPP07] which can support metavariables. They do not discuss the issues of type checking in the presence of unsolved constraints, but it is reasonable to believe that our work could be carried over to their modal type theory.

In this chapter we present a type checking algorithm which produces only well-typed constraints for a logical framework extended with metavariables. The main idea is, for a type-checking problem $t : C$, to produce a well-typed approximation t' of t. Whenever we need to verify $s : B$, for a subterm $s : A$ of t, where we cannot yet solve the constraint $A = B$, we replace the subterm s by a *guarded constant* p of type B. This constant p will compute to s only when the constraint $A = B$ has been solved. The approximated term t' is in a trivial way well-typed in the logical framework without metavariables. In the example above the type $(x : F\ ?) \to F\ (\neg\ x)$ will be replaced by $(x : F\ ?) \to F\ (p\ x)$ where $p\ x : \mathit{Bool}$ will compute to $\neg\ x$ when the metavariable is replaced with the term true.

The algorithm is greatly inspired by the treatment of metavariables in Epigram [McB07], as described to us by McBride [McB06].

Our main application for this work is implicit syntax [Pol90] which allows for a more compact and readable representation of terms. Necula and Lee [NL98] show that terms where type information is omitted is more efficient to validate than type checking the complete proof term. This results rely on the fact that no type checking is required when instantiating metavariables. Their work differs from ours in that they consider a weaker framework where the constraint solving is guaranteed to succeed. The algorithm we present has been implemented and we have made experiments with examples of several thousand of metavariables, which shows that our algorithm scales up to at least medium sized problems.

3.2 The underlying logic MLF

In this chapter we do not consider the full UTT_Σ type theory, but we use the simpler theory of Martin-Löf's logical framework [NPS00] as the underlying

logic. In Section 3.7 we discuss the issues involved in extending this work to UTT_Σ and definitions by pattern matching.

Syntax The syntax of **MLF** is given by the following grammar.

$$\begin{array}{lcll} A, B & ::= & \mathsf{Set} \mid s \mid (x : A) \to A & \textit{types} \\ s, t & ::= & x \mid c \mid s\,t \mid \lambda x.M & \textit{terms} \\ \Gamma, \Delta & ::= & \varepsilon \mid \Gamma, x : A & \textit{contexts} \\ \Sigma & ::= & \varepsilon \mid \Sigma, c : A \mid \Sigma, c : A = s & \textit{signatures} \end{array}$$

We adopt the same syntactic conventions as for UTT_Σ (see Section 1.3). The signature contains axioms and non-recursive definitions.

Judgements The type system of **MLF** is presented in six mutually dependent judgement forms.

$\vdash_\Sigma$	Σ is a valid signature
$\Gamma \vdash_\Sigma \mathbf{valid}$	Γ is a valid context
$\Gamma \vdash_\Sigma A\ \mathbf{type}$	A is a valid type in Γ
$\Gamma \vdash_\Sigma M : A$	s has type A in Γ
$\Gamma \vdash_\Sigma A = B$	A and B are convertible types in Γ
$\Gamma \vdash_\Sigma M = N : A$	s and t are convertible terms of type A in Γ

The typing rules follows standard presentations of type theory [NPS00] and can be obtained by suitably restricting the typing rules for UTT_Σ from Section 1.3.

Properties When proving the properties of the type checking algorithm in Section 3.3 we take the following properties of **MLF** for granted.

Lemma 3.2.1 (Uniqueness of types)**.**

$$\frac{\Gamma \vdash c\,\bar{s} : A \quad \Gamma \vdash c\,\bar{s} : B}{\Gamma \vdash A = B}$$

Lemma 3.2.2 (Constructor inversion)**.**

$$\frac{\Gamma \vdash c : \Delta \to B \quad \Gamma \vdash c\,\bar{s} : B'}{\Gamma \vdash \bar{s} : \Delta}$$

Lemma 3.2.3 (Substitution lemma)**.**

$$\frac{\Gamma \vdash s : B \quad x : A \in \Gamma \quad \Gamma \vdash t : A}{\Gamma \vdash s[x := t] : B[x := t]}$$

$$\frac{\Gamma \vdash B \ \mathbf{type} \quad x : A \in \Gamma \quad \Gamma \vdash s : A}{\Gamma \vdash B[x := s] \ \mathbf{type}}$$

Lemma 3.2.4 (Subject reduction)**.**

$$\frac{\Gamma \vdash s : A \quad s \rightarrow_{whnf} s'}{\Gamma \vdash s' : A}$$

Lemma 3.2.5 (Strengthening)**.**

$$\frac{\Gamma, x : A \vdash s : B \quad x \notin \mathsf{FV}(s) \cup \mathsf{FV}(B)}{\Gamma \vdash s : B}$$

$$\frac{\Gamma, x : A \vdash B \ \mathbf{type} \quad x \notin \mathsf{FV}(B)}{\Gamma \vdash B \ \mathbf{type}}$$

3.3 The type checking algorithm

In this section we present the type checking algorithm for **MLF** extended with metavariables. We refer to this system as $\mathbf{MLF_C}$. Note that the syntax of terms is the same in $\mathbf{MLF_C}$ as in **MLF**. The only thing we change is the syntax of signatures to include guarded constants. We also introduce a new syntactic category for user expressions:

$$\begin{array}{lcl} C & ::= & \Gamma \vdash A = B \mid \Gamma \vdash s = t : A \mid \Gamma \vdash \bar{s} = \bar{t} : \Delta \\ \Sigma & ::= & \ldots \mid \Sigma, p : A = s \ \mathbf{when}\ \mathcal{C} \\ e & ::= & x \mid c \mid \_ \mid \lambda x.e \mid e\, e \mid \mathsf{Set} \mid (x : e) \rightarrow e \end{array}$$

The input to the type checking algorithm is a user expression which could represent either a type or a term. Apart from the usual constructions user expressions can also contain _, representing a metavariable. During type checking user expressions are translated into **MLF** terms where metavariables are represented as fresh constants.

A constraint C is an equation which has been postponed because not enough information was available about the metavariables. Since our conversion checking algorithm is typed the constraints must also be typed. The constraints show up in the signature as guards to guarded constants. We

$$\begin{array}{lll}
\langle\Sigma\rangle\ \mathsf{Lookup}(c : A) \Longrightarrow \langle\Sigma\rangle & \text{if} & c : A \in \Sigma \\
\langle\Sigma\rangle\ \mathsf{AddMeta}(\alpha : A) \Longrightarrow \langle\Sigma, \alpha : A\rangle & \text{if} & \alpha \notin \Sigma \\
\langle\Sigma\rangle\ \alpha := s \Longrightarrow \langle\Sigma_1, \alpha : A = s, \Sigma_2\rangle & \text{if} & \Sigma = \Sigma_1, \alpha : A, \Sigma_2
\end{array}$$

$$\langle\Sigma\rangle\ \mathsf{AddConst}(p : A = s\ \mathbf{when}\ \mathcal{C}) \Longrightarrow \langle\Sigma, p : A = s\ \mathbf{when}\ \mathcal{C}\rangle \quad \text{if} \quad p \notin \Sigma$$

$$\langle\Sigma\rangle\ \mathsf{InScope}_\alpha(s) \Longrightarrow \langle\Sigma\rangle \quad \text{if} \quad \Sigma = \Sigma_1, \alpha : A, \Sigma_2 \text{ and } c \in s \text{ implies } c \in \Sigma_1$$

Figure 3.1: Operations on the signature

write $p : A = s$ **when** $\mathcal{C}$ for a guarded constant p of type A, with candidate value s, and guard $\mathcal{C}$. We have the computation rule that p computes to s when $\mathcal{C}$ is the empty set.

We use the naming convention that lowercase greek letters $\alpha, \beta, \ldots$ stand for constants representing metavariables and p and q for guarded constants.

3.3.1 Operations on the signature

All rules work on a signature Σ, containing previously defined constants, metavariables, and guarded constants. In other words, we can write all judgements on the form $\langle\Sigma\rangle\ J \Longrightarrow \langle\Sigma'\rangle$. To make the rules easier to read we first define a set of operations reading and modifying the signature and when presenting the algorithm simply write J for the judgement above. In rules with multiple premisses the signature is threaded top-down, left-to-right. For instance,

$$\frac{P_1 \qquad P_2 \qquad P_3}{J} \quad \text{is short-hand for} \quad \frac{\langle\Sigma_1\rangle\ P_1 \Longrightarrow \langle\Sigma_2\rangle \qquad \langle\Sigma_2\rangle\ P_2 \Longrightarrow \langle\Sigma_3\rangle \qquad \langle\Sigma_3\rangle\ P_3 \Longrightarrow \langle\Sigma_4\rangle}{\langle\Sigma_1\rangle\ J \Longrightarrow \langle\Sigma_4\rangle}$$

We introduce an operation $\mathsf{Lookup}(c : A)$ to look up the type of a constant in the signature. To manipulate metavariables we introduce: $\mathsf{AddMeta}(\alpha : A)$ which adds a new metavariable α of type A to the signature, and $\alpha := s$ which instantiates α to s. For guarded constants we add the operation $\mathsf{AddConst}(p : A = s\ \mathbf{when}\ \mathcal{C})$ to add a new guarded constant to the signature. In Section 3.3.2 we explain the rules for solving the constraints of a guarded constant. We also introduce an operation $\mathsf{InScope}_\alpha(s)$ to check that s is in scope at the definition site of α (to ensure that α can be instantiated to s). Detailed definitions of the operations can be found in Figure 3.1.

3.3.2 The algorithm

Next we present the type checking algorithm. We use a bidirectional algorithm, consisting of the following main judgement forms.

$\Gamma \vdash e\ \mathbf{type} \rightsquigarrow A$	well-formed types
$\Gamma \vdash e \uparrow A \rightsquigarrow s$	type checking
$\Gamma \vdash e \downarrow A \rightsquigarrow s$	type inference
$\Gamma \vdash A \simeq B \rightsquigarrow \mathcal{C}$	type conversion
$\Gamma \vdash s \simeq t \uparrow A \rightsquigarrow \mathcal{C}$	term conversion

The rules for well-formed types and type checking and inference take a user expression and produce a type or term which is a well-typed approximation of the user expression in **MLF**. Conversion checking produces a set of unsolved constraints which needs to be solved for the judgement to be true in **MLF**.

The algorithm maintains the following invariants: signatures and contexts are always well-formed, when checking $\Gamma \vdash e \uparrow A \rightsquigarrow s$ we have $\Gamma \vdash A\ \mathbf{type}$ in **MLF**, and when checking $\Gamma \vdash s \simeq t \uparrow A \rightsquigarrow \mathcal{C}$ we have $\Gamma \vdash s : A$ and $\Gamma \vdash t : A$ in **MLF**.

When checking conversion we also need the following judgement forms.

$\Gamma \vdash s \simeq' t \uparrow A \rightsquigarrow \mathcal{C}$	conversion of weak head normal forms
$\Gamma \vdash \bar{s} \simeq \bar{t} \uparrow \Delta \rightsquigarrow \mathcal{C}$	conversion of sequences of terms

Type checking with dependent types involves normalising arbitrary (type correct) terms, so we need to know how to normalise terms in an $\mathbf{MLF_C}$ signature. We do this by translating the signature to **MLF** and performing the normalisation in **MLF**.

Definition 3.3.1. *Given an* $\mathbf{MLF_C}$ *signature* Σ *we define its* **MLF** *restriction* $|\Sigma|$ *by replacing guarded constants with normal constants, replacing* $p : A = s$ **when** $\mathcal{C}$ *by* $p : A = s$ *if* $\mathcal{C}$ *is empty, and* $p : A$ *otherwise.*

The correctness of the type checking algorithm relies on the invariant that when $\langle\Sigma\rangle\ \Gamma \vdash e \uparrow A \rightsquigarrow s \Longrightarrow \langle\Sigma'\rangle$, we have $\Gamma \vdash_{|\Sigma'|} s : A$ (see Theorem 3.5.5).

We write $\langle\Sigma\rangle\ s \rightarrow_{whnf} s' \Longrightarrow \langle\Sigma\rangle$ if s' is the weak head normal form of s in $|\Sigma|$. Similarly $s \rightarrow_{nf} s'$ means that s' is the normal form of s.

Type checking rules

The rules for checking well-formedness of types are given in Figure 3.2 and the rules for type inference are presented in Figure 3.3. The type checking

$$\boxed{\Gamma \vdash e \ \mathbf{type} \ \rightsquigarrow A}$$

$$\frac{}{\Gamma \vdash \mathsf{Set} \ \mathbf{type} \ \rightsquigarrow \mathsf{Set}} \qquad \frac{\Gamma \vdash e_1 \ \mathbf{type} \ \rightsquigarrow A \qquad \Gamma, x : A \vdash e_2 \ \mathbf{type} \ \rightsquigarrow B}{\Gamma \vdash (x : e_1) \to e_2 \ \mathbf{type} \ \rightsquigarrow (x : A) \to B}$$

$$\frac{\Gamma \vdash e \uparrow \mathsf{Set} \rightsquigarrow s}{\Gamma \vdash e \ \mathbf{type} \ \rightsquigarrow s}$$

Figure 3.2: Checking for well-formed types

$$\boxed{\Gamma \vdash e \downarrow A \rightsquigarrow s}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \downarrow A \rightsquigarrow x} \qquad \frac{\mathsf{Lookup}(c : A)}{\Gamma \vdash c \downarrow A \rightsquigarrow c}$$

$$\frac{\Gamma \vdash e_1 \downarrow (x : A) \to B \rightsquigarrow s \qquad \Gamma \vdash e_2 \uparrow A \rightsquigarrow t}{\Gamma \vdash e_1 \, e_2 \downarrow B[x := N] \rightsquigarrow s \, t}$$

Figure 3.3: Type inference rules

$$\boxed{\Gamma \vdash e \uparrow A \rightsquigarrow s}$$

$$\frac{\Gamma, x : A \vdash e \uparrow B \rightsquigarrow s}{\Gamma \vdash \lambda x.e \uparrow (x : A) \to B \rightsquigarrow \lambda x.M} \qquad \frac{\mathsf{AddMeta}(\alpha : \Gamma \to A)}{\Gamma \vdash \_ \uparrow A \rightsquigarrow \alpha \, \Gamma}$$

$$\frac{\Gamma \vdash e \downarrow B \rightsquigarrow s \qquad \Gamma \vdash A \simeq B \rightsquigarrow \emptyset}{\Gamma \vdash e \uparrow A \rightsquigarrow s}$$

$$\frac{\Gamma \vdash A \simeq B \rightsquigarrow \mathcal{C} \neq \emptyset \qquad \begin{array}{c} \Gamma \vdash e \downarrow B \rightsquigarrow s \\ \mathsf{AddConst}(p : \Gamma \to A = \lambda \Gamma.s \ \mathbf{when} \ \mathcal{C}) \end{array}}{\Gamma \vdash e \uparrow A \rightsquigarrow p \, \Gamma}$$

Figure 3.4: Type checking rules

$$\boxed{\Gamma \vdash A \simeq B \rightsquigarrow \mathcal{C}}$$

$$\frac{}{\Gamma \vdash \mathsf{Set} \simeq \mathsf{Set} \rightsquigarrow \emptyset} \qquad \frac{\Gamma \vdash s \simeq t \uparrow \mathsf{Set} \rightsquigarrow \mathcal{C}}{\Gamma \vdash s \simeq t \rightsquigarrow \mathcal{C}}$$

$$\frac{\Gamma \vdash A_1 \simeq A_2 \rightsquigarrow \emptyset \qquad \Gamma,\, x : A_1 \vdash B_1 \simeq B_2 \rightsquigarrow \mathcal{C}}{\Gamma \vdash (x : A_1) \to B_1 \simeq (x : A_2) \to B_2 \rightsquigarrow \mathcal{C}}$$

$$\frac{\begin{array}{c}\Gamma \vdash A_1 \simeq A_2 \rightsquigarrow \mathcal{C}, \quad \mathcal{C} \neq \emptyset \\ \mathsf{AddConst}(p : \Gamma \to A_1 \to A_2 = \lambda \Gamma\, x.x \ \mathbf{when}\ \mathcal{C}) \\ \Gamma,\, x : A_1 \vdash B_1 \simeq B_2[x := p\ \Gamma\ x] \rightsquigarrow \mathcal{C}'\end{array}}{\Gamma \vdash (x : A_1) \to B_1 \simeq (x : A_2) \to B_2 \rightsquigarrow \mathcal{C} \cup \mathcal{C}'}$$

Figure 3.5: Type conversion rules

rules given in Figure 3.4 are more interesting, in particular the rules for checking a metavariable and the two conversion rules.

When type checking a user metavariable we create a fresh metavariable, add it to the signature and return it. Since metavariables are part of the signature they have to be lifted to the top-level.

We have two versions of the conversion rule. The first corresponds to the normal conversion rule and applies when no constraints are generated. The interesting case is when we cannot safely conclude that $A = B$, in which case we introduce a fresh guarded constant. As metavariables, guarded constants are lifted to the top-level.

Conversion rules

The rules for checking conversion of types are given in Figure 3.5. When checking conversion of two function types, an interesting question is what to do when comparing the domains gives rise to constraints. To ensure the correctness of the algorithm we need to maintain the invariant that when we check $\vdash A \simeq B \rightsquigarrow \mathcal{C}$ we have $\vdash A$ **type** and $\vdash B$ **type**. Thus if we do not know whether $A_1 = A_2$ it is not correct to check $x : A_1 \vdash B_1 \simeq B_2 \rightsquigarrow \mathcal{C}'$ since B_2 is not well-formed in the context $x : A_1$. To solve the problem we substitute a guarded constant $p\,x$ for x in B_2, where $p\,x$ reduces to x when A_1 and A_2 are convertible.

$$\boxed{\Gamma \vdash s \simeq t \uparrow A \rightsquigarrow \mathcal{C}}$$

$$\frac{\Gamma, x : A \vdash s\, x \simeq t\, x \uparrow B \rightsquigarrow \mathcal{C}}{\Gamma \vdash s \simeq t \uparrow (x : A) \to B \rightsquigarrow \mathcal{C}}$$

$$\frac{s \to_{whnf} s' \qquad t \to_{whnf} t' \qquad \Gamma \vdash s' \simeq' t' \uparrow A \rightsquigarrow \mathcal{C}}{\Gamma \vdash s \simeq t \uparrow A \rightsquigarrow \mathcal{C}}$$

Figure 3.6: Term conversion rules

$$\boxed{\Gamma \vdash s \simeq' t \uparrow A \rightsquigarrow \mathcal{C}}$$

$$\frac{h : \Delta \to A \qquad \Gamma \vdash \bar{s} \simeq \bar{t} \uparrow \Delta \rightsquigarrow \mathcal{C}}{\Gamma \vdash h\ \bar{s} \simeq' h\ \bar{t} \uparrow A[\Delta := \bar{s}] \rightsquigarrow \mathcal{C}} \qquad \frac{}{\Gamma \vdash p\ \bar{s} \simeq' t \uparrow A \rightsquigarrow \{\Gamma \vdash p\ \bar{s} = t : A\}}$$

$$\frac{\bar{x}\ \textit{distinct} \qquad s \to_{nf} s' \qquad \mathsf{FV}(s') \subseteq \bar{x} \qquad \mathsf{InScope}_{\alpha}(\lambda \bar{x}.s') \qquad \alpha := \lambda \bar{x}.s'}{\Gamma \vdash \alpha\ \bar{x} \simeq' s \uparrow A \rightsquigarrow \emptyset}$$

Figure 3.7: Conversion rules for weak head normal forms.

Term conversion rules

Checking conversion of terms is done on weak head normal forms. The only rule that is applied before weak head normalisation is the η-rule shown in Figure 3.6. In **MLF** function types are not terms so a metavariable can never be instantiated to a function type. When extending the algorithm to UTT_{Σ}, where this is the case, we have to check if the type is a metavariable, and if so postpone the constraint, since we do not know whether or not the η-rule should be applied (see Section 3.7.2).

The conversion rules for weak head normal forms are shown in Figure 3.7. The weak head normal forms we compare will be of atomic type and so they are of the form $h\,\bar{s}$ where the head h is a variable, constant, metavariable, or guarded constant. If both terms have the same variable or constant head $h : \Delta \to A$ we compare the arguments in Δ. Note that it is not necessary to check that the given type is indeed $A[\Delta := \bar{s}]$—this is guaranteed by the fact that the constraint is well-typed.

If the heads are different constants or variables conversion checking fails.

$$\boxed{\Gamma \vdash \bar{s} \simeq \bar{t} \uparrow \Delta \rightsquigarrow \mathcal{C}}$$

$$\frac{\Gamma \vdash s \simeq t \uparrow A \rightsquigarrow \emptyset \qquad \Gamma \vdash \bar{s} \simeq \bar{t} \uparrow \Delta[x := s] \rightsquigarrow \mathcal{C}}{\Gamma \vdash s; \bar{s} \simeq t; \bar{t} \uparrow (x : A)\Delta \rightsquigarrow \mathcal{C}}$$

$$\frac{\Gamma \vdash s \simeq t \uparrow A \rightsquigarrow \mathcal{C} \neq \emptyset \qquad x \in \mathsf{FV}(\Delta)}{\Gamma \vdash s; \bar{s} \simeq t; \bar{t} \uparrow (x : A)\Delta \rightsquigarrow \{\Gamma \vdash s; \bar{s} = t; \bar{t} : (x : A)\Delta\}}$$

$$\frac{\Gamma \vdash s \simeq t \uparrow A \rightsquigarrow \mathcal{C}_1 \neq \emptyset \qquad \Gamma \vdash \bar{s} \simeq \bar{t} \uparrow \Delta \rightsquigarrow \mathcal{C}_2 \qquad x \notin \mathsf{FV}(\Delta)}{\Gamma \vdash s; \bar{s} \simeq t; \bar{t} \uparrow (x : A)\Delta \rightsquigarrow \mathcal{C}_1 \cup \mathcal{C}_2}$$

Figure 3.8: Conversion checking sequences of terms

If one of the heads is a guarded constant we give up and return the problem as a constraint.

If one of the heads is a metavariable we use a restricted form of pattern unification, but we believe that our correctness proof can be extended to more powerful unification algorithms, for example [Dow01, DHK95, Mil91, Nip93, Pfe91]. The crucial step is to prove that metavariable instantiations are well-typed. In the examples we have studied, using metavariables for implicit arguments, this simpler form of unification seems to be sufficient.

Given the problem $\alpha\,\bar{x} = s$ we would like to instantiate α to $\lambda\bar{x}.s$. This is only correct if $\bar{x}$ are distinct variables, s does not contain any variables other than $\bar{x}$, and any constants referred to by s are in scope at the declaration site of α[1]. Now s might refer to metavariables introduced after α but which have been instantiated. For this reason we normalise s to s' and try to instantiate α to $\lambda\bar{x}.s'$. A possible optimisation might be to only normalise if s contains out-of-scope constants or variables. A possible improvement might be to allow consecutive metavariables in the signature to be permuted, and so allow a metavariable to be instantiated with a term containing metavariables defined later in the signature, but not after any proper constants. It is unclear how much this would buy us, since in most cases we would expect to be able to solve the later metavariables first.

If any of the premisses in this rule fail or α is not applied only to variables, we return the constraint as it is.

When checking conversion of argument lists (Figure 3.8), the interest-

[1]Note that scope checking subsumes the usual occurs check, since constants are non-recursive.

ing case is when comparing the first arguments results in some unsolved constraints. If the value of the first argument is used in the types of later arguments ($x \in \mathsf{FV}(\Delta)$) we have to stop and produce a constraint since the types of $\bar{s}$ and $\bar{t}$ differ. If on the other hand the types of later arguments are independent of the value of the first argument, we can proceed and compare them without knowing whether the first arguments are convertible.

Constraint Solving

So far, we have not looked at when or how the guards of a constant are simplified or solved. In principle this can be done at any time, for instance as a separate phase after type checking. In practice, however, it might be a better idea to interleave constraint solving and type checking. In Section 3.5 we prove that this can be done safely. Constraint solving amounts to rechecking the guard of a constant and replacing it by the resulting constraints.

3.4 Examples

In this section we look at a few examples which illustrate the workings of the type checker.

A simple example

First let us look at a very simple example. Consider the signature Σ given by

$$\begin{array}{lll} \mathit{Nat} & : \mathsf{Set} \\ 0 & : \mathit{Nat} \\ \mathit{id} & : (A : \mathsf{Set}) \to A \to A \\ & = \lambda A\ x.\ x \\ \alpha & : \mathsf{Set} \end{array}$$

containing a set Nat with an element 0, a polymorphic identity function id, and a metavariable α of type Set. Now we want to compute s such that

$$\vdash \mathit{id}\ \_\ 0 \uparrow \alpha \rightsquigarrow s$$

To do this one of the conversion rules has to be applied, so the type checker first infers the type of $\mathit{id}\ \_\ 0$.

$$\dfrac{\vdash \mathit{id} \downarrow (A : \mathsf{Set}) \to A \to A \rightsquigarrow \mathit{id} \qquad \vdash \_ \uparrow \mathsf{Set} \rightsquigarrow \beta \qquad \dfrac{\vdash 0 \downarrow \mathit{Nat} \rightsquigarrow 0 \qquad \beta := \mathit{Nat}}{\vdash 0 \uparrow \beta \rightsquigarrow 0}}{\vdash \mathit{id}\ \_\ 0 \downarrow \beta \rightsquigarrow \mathit{id}\ \beta\ 0}$$

The inferred type β is then compared against the expected type α, resulting in the instantiation $\alpha := Nat$. The final signature is

$$\begin{aligned} Nat &: \mathsf{Set} \\ 0 &: Nat \\ id &: (A : \mathsf{Set}) \to A \to A = \lambda A\ x.\ x \\ \alpha &: \mathsf{Set} = Nat \\ \beta &: \mathsf{Set} = Nat \end{aligned}$$

and we have $s = id\ \beta\ 0$. Note that it is important to look up the values of instantiated metavariables—it would not be correct to instantiate α to β, since β is not in scope at the point where α is declared (α appears before β in the signature).

An example with guarded constants

In the previous example all constraints could be solved immediately so no guarded constants had to be introduced. Now let us look at an example with guarded constants. Consider the signature of natural numbers with a case principle:

$$\begin{aligned} Nat &: \mathsf{Set} \\ 0 &: Nat \\ suc &: Nat \to Nat, \\ caseNat &: (P : Nat \to \mathsf{Set}) \to P\ 0 \to \\ &\quad ((n : Nat) \to P\ (suc\ n)) \to \\ &\quad (n : Nat) \to P\ n \end{aligned}$$

In this signature we want to check that $caseNat\ \_\ 0\ (\lambda n.\ n)$ has type $Nat \to Nat$. The first thing that happens is that the arguments to $caseNat$ are checked against their expected types. Checking $\_$ against $Nat \to \mathsf{Set}$ introduces a fresh metavariable

$$\alpha : Nat \to \mathsf{Set}$$

Next the inferred type of 0 is checked against the expected type $\alpha\ 0$. This produces an unsolved constraint $\alpha\ 0 = Nat$, so a guarded constant is introduced:

$$p : \alpha\ 0 = 0\ \textbf{when}\ \alpha\ 0 = Nat$$

Similarly, the third argument introduces a guarded constant.

$$q : (n : Nat) \to \alpha\ (suc\ n) = \lambda n.\ n\ \textbf{when}\ (n : Nat) \vdash \alpha\ (suc\ n) = Nat$$

The resulting type correct approximation is $caseNat\ \alpha\ p\ (\lambda n.\ q\ n)$ of type $(n{:}Nat) \to \alpha\ n$. This type is compared against the expected type $Nat \to Nat$ giving rise to the constraint $\alpha\ n = Nat$ which is solvable with $\alpha = \lambda n.\ Nat$. Once α is instantiated we can solve the guards on p and q and subsequently reduce $caseNat\ \alpha\ p\ (\lambda n.\ q\ n)$ to $caseNat\ (\lambda n.\ Nat)\ 0\ (\lambda n.\ n) : Nat \to Nat$.

What could go wrong?

So far we have only looked at type correct examples, where nothing bad would have happened if we had not introduced guarded constants when we did. The following example shows how things can go wrong. Take the signature $Nat : \mathsf{Set}, 0 : Nat$. Now add the perfectly well-typed identity function *coerce*:

$$coerce : (F : Nat \to \mathsf{Set}) \to F\ 0 \to F\ 0 = \lambda F\ x.\ x$$

For any well-typed term $t : B$ and type A, *coerce* _ t will successfully check against A, resulting in the constraints $\alpha\ 0 = B$ and $A = \alpha\ 0$, none of which can be solved. If we did not introduce guarded constants *coerce* _ t would reduce to t and hence we could use *coerce* to give an arbitrary type to a term. For instance we can type[2]

$$\begin{aligned} \omega &: (Nat \to Nat) \to Nat = \lambda x.\ x\ (coerce\ \_\ x) \\ \Omega &: Nat \qquad\qquad\qquad\quad = \omega\ (coerce\ \_\ \omega) \end{aligned}$$

where, without guarded constants, Ω would reduce to the non-normalising λ-term $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$. With our algorithm new guarded constants are introduced for for the argument to *coerce* and for the application of coerce. So the type correct approximation of Ω would be $\omega\ p$ where

$$\begin{aligned} p &= coerce\ \alpha\ q \ \textbf{when}\ \alpha\ 0 = Nat \to Nat \\ q &= \omega \qquad\quad \textbf{when}\ (Nat \to Nat) \to Nat = \alpha\ 0 \end{aligned}$$

3.5 Proof of correctness

The correctness of the algorithm relies on the fact that we only compute with well-typed terms. This guarantees the existence of normal forms and hence ensures the termination of the type checking algorithm.

The proof will be done in two stages: first we prove soundness in the absence of constraint solving, and then we prove that constraint solving is sound.

3.5.1 Soundness without constraint solving

There are a number of things we need to prove: that type checking preserves well-formed signatures, that it produces well-typed terms, that conversion checking is sound, and that new signatures respect the old signatures. Unfortunately these properties are all interdependent, so we cannot prove them separately.

[2]This only type checks if we allow metavariables to be instantiated to function types, which is not the case in **MLF**. See Section 3.7.2 for a discussion on how to extend the algorithm to handle this

Definition 3.5.1 (Signature extension)**.** *We say that* Σ' extends Σ *if for any* **MLF** *judgement* J, $\vdash_\Sigma J$ *implies* $\vdash_{\Sigma'} J$.

Note that this definition admits both simple extensions–adding a new constant–and refinement, where we give a definition to a constant. This is expressed by the following two lemmas.

Lemma 3.5.2 (Signature weakening)**.** *If* $\vdash_{\Sigma, c:A}$ *then* $\Sigma, c : A$ *extends* Σ.

Lemma 3.5.3 (Signature refinement)**.** *Giving a definition to a constant in a signature is an extension of the signature. More precisely, if*

- $\vdash_{\Sigma_1} s : A$
- $\Sigma \;=\; \Sigma_1, c : A, \Sigma_2$
- $\Sigma' \;=\; \Sigma_1, c : A = s, \Sigma_2$

then Σ' *extends* Σ.

Proofs. In both lemmas any derivation using Σ is immediately valid also with Σ'. □

To express the soundness of conversion checking we need to define when a constraint is well-formed. Note that this is not the same as being true. For instance, $\Gamma \vdash \mathit{Nat} = \mathit{Bool}$ is a well-formed constraint given $\mathit{Nat} : \mathsf{Set}$ and $\mathit{Bool} : \mathsf{Set}$ in the signature.

Definition 3.5.4 (Well-formed constraint)**.** *A constraint* C *is well-formed in an* **MLF** *signature* Σ, *written* $\vdash_\Sigma C$ **ok**, *if the terms (or types) under consideration are well-typed.*

$$\frac{\Gamma \vdash_\Sigma A \textbf{ type} \quad \Gamma \vdash_\Sigma B \textbf{ type}}{\vdash_\Sigma \Gamma \vdash A = B \textbf{ ok}} \qquad \frac{\Gamma \vdash_\Sigma s : A \quad \Gamma \vdash_\Sigma t : A}{\vdash_\Sigma \Gamma \vdash s = t : A \textbf{ ok}}$$

Now we are ready to state the soundness of the type checking algorithm in the absence of constraint solving.

Theorem 3.5.5 (Soundness of type checking)**.** *Type checking produces well-typed terms, conversion checking produces well-formed constraints and if no constraints are produced, the conversion is valid in* **MLF**. *Also, all rules*

produce well-formed extensions of the signature. More precisely, the following rules are admissible:

$$\frac{\langle\Sigma\rangle\ \Gamma \vdash e\ \mathbf{type} \rightsquigarrow A \Longrightarrow \langle\Sigma'\rangle \qquad \Gamma \vdash_{|\Sigma|} \mathbf{valid}}{\Sigma'\ \mathit{extends}\ \Sigma \quad \wedge \quad \Gamma \vdash_{|\Sigma'|} A\ \mathbf{type}}$$

$$\frac{\langle\Sigma\rangle\ \Gamma \vdash e \uparrow A \rightsquigarrow s \Longrightarrow \langle\Sigma'\rangle \qquad \Gamma \vdash_{|\Sigma|} A\ \mathbf{type}}{\Sigma'\ \mathit{extends}\ \Sigma \quad \wedge \quad \Gamma \vdash_{|\Sigma'|} s : A}$$

$$\frac{\langle\Sigma\rangle\ \Gamma \vdash e \downarrow A \rightsquigarrow s \Longrightarrow \langle\Sigma'\rangle \qquad \Gamma \vdash_{|\Sigma|} \mathbf{valid}}{\Sigma'\ \mathit{extends}\ \Sigma \quad \wedge \quad \Gamma \vdash_{|\Sigma'|} s : A}$$

$$\frac{\langle\Sigma\rangle\ \Gamma \vdash A \simeq B \rightsquigarrow \mathcal{C} \Longrightarrow \langle\Sigma'\rangle \qquad \Gamma \vdash_{|\Sigma|} A\ \mathbf{type} \qquad \Gamma \vdash_{|\Sigma|} B\ \mathbf{type}}{\Sigma'\ \mathit{extends}\ \Sigma \quad \wedge \quad \vdash_{|\Sigma'|} \mathcal{C}\ \mathbf{ok} \quad \wedge \quad (\mathcal{C} = \emptyset \implies \Gamma \vdash_{|\Sigma'|} A = B)}$$

$$\frac{\langle\Sigma\rangle\ \Gamma \vdash s \simeq t \uparrow A \rightsquigarrow \mathcal{C} \Longrightarrow \langle\Sigma'\rangle \qquad \Gamma \vdash_{|\Sigma|} s : A \qquad \Gamma \vdash_{|\Sigma|} t : A}{\Sigma'\ \mathit{extends}\ \Sigma \quad \wedge \quad \vdash_{|\Sigma'|} \mathcal{C}\ \mathbf{ok} \quad \wedge \quad (\mathcal{C} = \emptyset \implies \Gamma \vdash_{|\Sigma'|} s = t : A)}$$

The statements for weak head normal form conversion ($\Gamma \vdash s \simeq' t \uparrow A \rightsquigarrow \mathcal{C}$) and term sequence conversion ($\Gamma \vdash \bar{s} \simeq \bar{t} \uparrow \Delta \rightsquigarrow \mathcal{C}$) are equivalent to that of term conversion.

Proof. By induction on the derivation. Some interesting cases:

- In the type conversion case for function spaces where the domains produce constraints, we have to use the substitution lemma (Lemma 3.2.3) and strengthening (Lemma 3.2.5).

- In the term conversion case where the terms are weak head normalised we need subject reduction for weak head normalisation (Lemma 3.2.4).

- When checking conversion of terms with the same head we need an inversion principle for application (Lemma 3.2.2).

- The most interesting case is the metavariable instantiation case, so let us spell that out in more detail.

  The instantiation rule does not produce any constraints, so the only thing we have to prove is that it constructs a valid extension of the signature. This follows from the signature refinement lemma (Lemma 3.5.3) which can be applied if we prove that if $\Sigma = \Sigma_1, \alpha : B', \Sigma_2$ then $\vdash_{|\Sigma_1|} \lambda\bar{x}.s : B'$.

We have $\Gamma \vdash_{|\Sigma|} \alpha\,\bar{x} : A$ so B' must have the form $(\bar{x} : \Delta) \to B$. By Lemma 3.2.2 we conclude that $\Gamma \vdash_{|\Sigma|} \bar{x} : \Delta$ and thus $\Gamma \vdash_{|\Sigma|} \alpha\,\bar{x} : B$. Then by Lemma 3.2.1 $\Gamma \vdash_{|\Sigma|} A = B$.

From $\Gamma \vdash_{|\Sigma|} s : A$ we get $\Gamma \vdash_{|\Sigma|} s : B$ and using Lemma 3.2.3 $\Gamma \vdash_{|\Sigma|} \lambda\bar{x}.s : \Delta \to B$. We know that $\Delta \to B$ is a closed type, and since $\mathsf{FV}(s) \subseteq \bar{x}$, $\lambda\bar{x}.s$ is also closed. Thus by strengthening (Lemma 3.2.5) $\vdash_{|\Sigma|} \lambda\bar{x}.s : \Delta \to B$. We have $\vdash_{|\Sigma_1|} \Delta \to B$ **type** and $\mathsf{InScope}_\alpha(s)$ so $\vdash_{|\Sigma_1|} \lambda\bar{x}.s : \Delta \to B$ which is what we set out to prove.

□

Since well-typed terms in **MLF** have normal forms we get the existence of normal forms for type checked terms and hence the type checking algorithm is terminating (the only part of the algorithm which is not structurally recursive is when we compute normal forms).

Corollary 3.5.6. *The type checking algorithm is terminating.*

Note that type checking terminates with one of three answers: *yes it is type correct*, *no it is not correct*, or *it might be correct if the metavariables are instantiated properly*. The algorithm is not complete, since finding correct instantiations to the metavariables is undecidable in the general case.

3.5.2 Soundness of constraint solving

In the previous section we proved type checking sound and decidable in the absence of constraint solving. We also mostly ignored the constraints, only requiring them to be well-formed. In this section we prove that the terms produced by the type checker stay well-typed under constraint solving. This is done by showing that constraint solving is a signature extension operator in the sense of Definition 3.5.1.

Previously we only ensured that the **MLF** restriction of the signature was well-formed. Now, since we are going to update and remove the constraints of guarded constants we have to strengthen the requirements and demand *consistent* signatures. A signature is consistent if the solution of a guard is a sufficient condition for the well-typedness of the definition it is guarding.

Definition 3.5.7 (Solved constraints)**.** *A set of constraints $\mathcal{C}$ is* solved *in a signature Σ if $\vdash_{|\Sigma|} \mathcal{C}$ and all guards in Σ have been solved as far as possible, i.e. for any non-empty guard $\mathcal{C}'$ in Σ it is not the case that $\vdash_{|\Sigma|} \mathcal{C}'$.*

Definition 3.5.8 (Ensures)**.** *A set of constraints $\mathcal{C}$* ensures *an* **MLF** *judgement J in a signature Σ if, for any extension Σ' of Σ in which $\mathcal{C}$ is solved it is the case that $\vdash_{|\Sigma'|} J$.*

Remark 3.5.9. *If $\mathcal{C}$ ensures J in Σ and Σ' extends Σ then $\mathcal{C}$ ensures J in Σ'.*

Note that in the case when Σ' invalidates $\mathcal{C}$ the remark is vacuous—a false constraint ensures anything.

Definition 3.5.10 (Consistent signature)**.** *A signature Σ is said to be* consistent *if for any p with Σ equal to Σ_1, $p : A = s$* **when** *$\mathcal{C}$, Σ_2 it is the case that $\mathcal{C}$ ensures $\vdash s : A$ in Σ_1.*

In order to prove that type checking preserves consistency, we first need to know that the constraints we produce are sound.

Lemma 3.5.11 (Soundness of generated constraints)**.** *The constraints generated during conversion checking ensures that the checked terms are convertible. For instance, if $\Gamma \vdash A \simeq B \rightsquigarrow \mathcal{C}$, then solving $\mathcal{C}$ guarantees that $\Gamma \vdash A = B$ in* **MLF***. More precisely,*

- $\Gamma \vdash_\Sigma A$ **type** $\wedge$ $\Gamma \vdash_\Sigma B$ **type** $\wedge$ $\langle\Sigma\rangle\ \Gamma \vdash A \simeq B \rightsquigarrow \mathcal{C} \Longrightarrow \langle\Sigma'\rangle$ $\Longrightarrow$ *$\mathcal{C}$ ensures $\Gamma \vdash A = B$ in Σ'*
- $\Gamma \vdash_\Sigma s : A \ \wedge\ \Gamma \vdash_\Sigma t : A \ \wedge\ \langle\Sigma\rangle\ \Gamma \vdash s \simeq t \uparrow A \rightsquigarrow \mathcal{C} \Longrightarrow \langle\Sigma'\rangle$ $\Longrightarrow$ *$\mathcal{C}$ ensures $\Gamma \vdash s = t : A$ in Σ'*

Proof. Again we highlight some interesting cases.

- The only non-trivial case is the case of conversion for function types where a new constant p is introduced. There we need to prove that for a signature Σ' which solves the constraints generated by comparing A_1 with A_2 and B_1 with $B_2[x := p\,\Gamma\,x]$ it holds that $\Gamma,\ x : A_1 \vdash_{|\Sigma'|} B_1 = B_2$ given that $\Gamma,\ x : A_1 \vdash_{|\Sigma'|} B_1 = B_2[x := p\,\Gamma\,x]$

  Since Σ_2 solves $A_1 \simeq A_2$ it has an empty guard for p, so $p\,\Gamma\,x$ reduces to x and we are done.

- In the case where $\mathcal{C}$ is known (for instance, in the rule for blocked terms), we can apply soundness of conversion checking (Theorem 3.5.5) to get $\vdash_{|\Sigma'|} \mathcal{C}$.

□

Lemma 3.5.12. *Refinement preserves consistent signatures. More precisely, if*

- $\vdash_{\Sigma_1} s : A$

- $\Sigma \;=\; \Sigma_1,\, c : A,\, \Sigma_2$
- $\Sigma' \;=\; \Sigma_1,\, c : A = s,\, \Sigma_2$
- Σ *is consistent*

then Σ' *is consistent.*

Proof. There are two cases to consider: refinement to the left and to the right of a guard. In the latter case the proof is trivial, and in the former case consistency follows from the fact that refinement extends a signature (Lemma 3.5.3). □

Lemma 3.5.13 (Type checking preserves consistency)**.** *Type checking and conversion checking preserves consistent signatures. More precisely,*

- $\langle\Sigma\rangle\ \Gamma \vdash e\ \mathbf{type} \rightsquigarrow A \Longrightarrow \langle\Sigma'\rangle\ \wedge\ \Gamma \vdash_{|\Sigma|} \mathbf{valid}\ \wedge\ \Sigma$ *is consistent* $\Longrightarrow\ \Sigma'$ *is consistent*
- $\langle\Sigma\rangle\ \Gamma \vdash e \uparrow A \rightsquigarrow s \Longrightarrow \langle\Sigma'\rangle\ \wedge\ \Gamma \vdash_{|\Sigma|} A\ \mathbf{type}\ \wedge\ \Sigma$ *is consistent* $\Longrightarrow\ \Sigma'$ *is consistent*
- $\langle\Sigma\rangle\ \Gamma \vdash e \downarrow A \rightsquigarrow s \Longrightarrow \langle\Sigma'\rangle\ \wedge\ \Gamma \vdash_{|\Sigma|} \mathbf{valid}\ \wedge\ \Sigma$ *is consistent* $\Longrightarrow\ \Sigma'$ *is consistent*
- $\langle\Sigma\rangle\ \Gamma \vdash A \simeq B \rightsquigarrow \mathcal{C} \Longrightarrow \langle\Sigma'\rangle\ \wedge\ \Gamma \vdash_{|\Sigma|} A\ \mathbf{type}\ \wedge\ \Gamma \vdash_{|\Sigma|} B\ \mathbf{type}$ $\wedge\ \Sigma$ *is consistent* $\Longrightarrow\ \Sigma'$ *is consistent*
- $\langle\Sigma\rangle\ \Gamma \vdash s \simeq t \uparrow A \rightsquigarrow \mathcal{C} \Longrightarrow \langle\Sigma'\rangle\ \wedge\ \Gamma \vdash_{|\Sigma|} s : A\ \wedge\ \Gamma \vdash_{|\Sigma|} t : A$ $\wedge\ \Sigma$ *is consistent* $\Longrightarrow\ \Sigma'$ *is consistent*

The statements for weak head normal form conversion ($\Gamma \vdash s \simeq' t \uparrow A \rightsquigarrow \mathcal{C}$) *and term sequence conversion* ($\Gamma \vdash \bar{s} \simeq \bar{t} \uparrow \Delta \rightsquigarrow \mathcal{C}$) *are equivalent to that of term conversion.*

Proof. By induction on the derivation. We only need to consider the cases where the signature changes. Adding a (non-guarded) constant trivially preserves consistency, instantiating a metavariable preserves consistency by Lemma 3.5.12. What remains is to check that new guarded constants are consistent. There are two cases: the conversion rule and conversion checking of function types. In both cases consistency follows from soundness of conversion checking (Lemma 3.5.11). □

Lemma 3.5.14 (Constraint solving is sound)**.** *If Σ is consistent and solving the constraints yields a signature Σ', then Σ' is consistent and Σ' extends Σ.*

Proof. Follows from Theorem 3.5.5, Lemma 3.5.11, and Lemma 3.5.13. □

From this follows that we can mix type checking and constraint solving freely, so we can add a constraint solving rule to the type checking algorithm. In order to obtain optimal approximations we have to solve constraints eagerly, i.e. as soon as a metavariable has been instantiated.

3.5.3 Relating user expressions and checked terms

An important property of the type checking algorithm is that the type correct terms produced correspond to the expressions being type checked. The correspondence is expressed by stating that the only operations the type checker is allowed when constructing a term is replacing an $\_$ by a term (*refinement*) and replacing a term by a guarded constant with an appropriate candidate value (*approximation*).

Definition 3.5.15 (Approximation)**.** *A term s* approximates *s' if s can be obtained by replacing subterms t of s' by guarded constants $p\,\bar{x}$ whose candidate values approximates t.*

Definition 3.5.16 (Refinement)**.** *A term s is a* refinement *of a user expression e if s can be obtained by replacing the $\_$ in e by concrete terms.*

Lemma 3.5.17. *If $\Gamma \vdash e \uparrow A \rightsquigarrow s$ then s approximates a refinement of e. This property is preserved when unfolding instantiated metavariables and guarded constants in s.*

Proof. By induction over the derivation. □

3.5.4 Main result

We now prove the main soundness theorem stating that if all metavariables are instantiated and all guards solved, then the term produced by the type checker (extended with constraint solving) is valid in the original signature after unfolding the definitions of the metavariables and guarded constants introduced during type checking.

Theorem 3.5.18 (Soundness of type checking)**.** *If Σ is a well-formed* **MLF** *signature and $\langle\Sigma\rangle\ \Gamma \vdash e \uparrow A \rightsquigarrow s \Longrightarrow \langle\Sigma'\rangle$, then if all metavariables have been instantiated and all guards are empty in Σ', then $\Gamma \vdash_\Sigma s\sigma : A$ where σ is the substitution unfolding the metavariables and constants in Σ'. Moreover, $s\sigma$ is a refinement of e.*

Proof. From soundness of type checking (Theorem 3.5.5) follows that $\Gamma \vdash_{|\Sigma'|} s : A$ and Lemma 3.5.13 and Lemma 3.5.14 give that Σ' is a consistent extension of Σ. Thus σ is well-typed and we have $\Gamma \vdash_{|\Sigma'|} s\sigma : A$. Since $s\sigma$ only uses constants from Σ we can strengthen the signature to obtain $\Gamma \vdash_{\Sigma} s\sigma : A$.

By Lemma 3.5.17 we have that $s\sigma$ approximates a refinement of e. Since $s\sigma$ does not contain any guarded constants it is a refinement of e. □

3.6 Implicit arguments

Once we have metavariables, adding implicit arguments to our logic is simply a matter of inserting metavariables at the right places. One way of doing this is described below.

We add an implicit function space $\{x : A\} \to B$ whose only purpose is to guide the insertion of metavariables—semantically there is no difference between the implicit function space and the ordinary function space. This is in contrast to, for instance, the implicit calculus of constructions [Miq01] where implicit function spaces are intersections rather than products. To see the difference consider the function *downFrom* which constructs the vector of the numbers $n - 1, \ldots, 0$:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Vec}\ (A\ :\ \mathit{Set})\ :\ \mathit{Nat}\ \to\ \mathsf{Set}\ \textbf{where} \\
\quad \varepsilon\ \ \ :\ \mathit{Vec}\ A\ \mathsf{zero} \\
\quad \_::\_\ :\ \{n\ :\ \mathit{Nat}\}\ \to\ A\ \to\ \mathit{Vec}\ A\ n\ \to\ \mathit{Vec}\ A\ (\mathsf{suc}\ n) \\
\mathit{downFrom}\ :\ \{n\ :\ \mathit{Nat}\}\ \to\ \mathit{Vec}\ \mathit{Nat}\ n \\
\mathit{downFrom}\ \{\mathsf{zero}\}\ \ =\ \varepsilon \\
\mathit{downFrom}\ \{\mathsf{suc}\ n\}\ =\ n\ ::\ \mathit{downFrom}\ n
\end{array}
$$

Here it is safe to leave n implicit, since it can be inferred from the goal type, but it is clearly not the case that n is computationally irrelevant.

We also add corresponding abstractions and applications: $\lambda\{x\}.\, s$, and $s\,\{t\}$. These, however, are optional and are inserted by the type checker if not given explicitly.

The new type checking and inference rules are the following.

$$\frac{\Gamma, x : A \vdash e \uparrow B \rightsquigarrow s}{\Gamma \vdash \lambda\{x\}.\, e \uparrow \{x : A\} \to B \rightsquigarrow \lambda\{x\}.\, s}$$

$$\frac{\Gamma, x : A \vdash e \uparrow B \rightsquigarrow s}{\Gamma \vdash e \uparrow \{x : A\} \to B \rightsquigarrow \lambda\{x\}.\, s} \quad e \neq \lambda\{x\}.\, e'$$

$$\frac{x : A \in \Gamma \qquad \Gamma \vdash A \mathbin{@} \bar{e} \downarrow B \rightsquigarrow \bar{s}}{\Gamma \vdash x\, \bar{e} \downarrow B \rightsquigarrow x\, \bar{s}} \qquad \frac{\mathsf{Lookup}(x : A) \qquad \Gamma \vdash A \mathbin{@} \bar{e} \downarrow B \rightsquigarrow \bar{s}}{\Gamma \vdash c\, \bar{e} \downarrow B \rightsquigarrow c\, \bar{s}}$$

As can be seen, when checking an expression against an implicit function type an implicit lambda is inserted if needed. To check a function application we introduce a new judgement form $\Gamma \vdash A \mathbin{@} \bar{e} \downarrow B \rightsquigarrow \bar{s}$ with the meaning that a function of type A can be applied to the arguments $\bar{e}$ resulting in a term of type B. The terms $\bar{s}$ are the type correct approximations of the arguments. The rules basically inserts metavariables into $\bar{e}$ whenever necessary, and otherwise checks that the expressions have the expected types. One thing to note is that is there are implicit function spaces left over at the end they are instantiated with metavariables. The rules are

$$\boxed{\Gamma \vdash A \mathbin{@} \bar{e} \downarrow B \rightsquigarrow \bar{s}}$$

$$\frac{\Gamma \vdash e \uparrow A \rightsquigarrow s \qquad \Gamma \vdash B[x := s] \mathbin{@} \bar{e} \downarrow B' \rightsquigarrow \bar{s}}{\Gamma \vdash (x : A) \to B \mathbin{@} e; \bar{e} \downarrow B' \rightsquigarrow s; \bar{s}}$$

$$\frac{\Gamma \vdash e \uparrow A \rightsquigarrow s \qquad \Gamma \vdash B[x := s] \mathbin{@} \bar{e} \downarrow B' \rightsquigarrow \bar{s}}{\Gamma \vdash \{x : A\} \to B \mathbin{@} \{e\}; \bar{e} \downarrow B' \rightsquigarrow s; \bar{s}}$$

$$\frac{\Gamma \vdash \{x : A\} \to B \mathbin{@} \{\_\}; e; \bar{e} \downarrow B' \rightsquigarrow \bar{s}}{\Gamma \vdash \{x : A\} \to B \mathbin{@} e; \bar{e} \downarrow B' \rightsquigarrow \bar{s}} \qquad \frac{\Gamma \vdash \{x : A\} \to B \mathbin{@} \{\_\} \downarrow B' \rightsquigarrow \bar{s}}{\Gamma \vdash \{x : A\} \to B \mathbin{@} \varepsilon \downarrow B' \rightsquigarrow \bar{s}}$$

$$\frac{}{\Gamma \vdash A \mathbin{@} \varepsilon \downarrow A \rightsquigarrow \varepsilon} \quad A \neq \{x : A_1\} \to A_2$$

3.7 Extending the underlying theory

The algorithm presented in this chapter works on the logical framework **MLF** extended with metavariables. This framework lacks a number of features

present in UTT_Σ and the Agda language. This section briefly discusses how to extend the algorithm to cope with the additional features. The implementation of Agda uses this extended algorithm.

3.7.1 Sigma types and the unit type

Adding Σ and $\mathsf{1}$ presents no great difficulties. An interesting aspect, however, is that it provides us with the possibility of solving metavariables by η-expansion. Given

$$\alpha \ : \ \Gamma \ \rightarrow \ (x \ : \ A) \ \times \ B$$

we can solve α with

$$\alpha \ := \ \lambda \Gamma . \, \langle \beta \ \Gamma , \ \gamma \ \Gamma \rangle$$

for fresh metavariables β and γ of type

$$\begin{array}{l} \beta \ : \ \Gamma \ \rightarrow \ A \\ \gamma \ : \ \Gamma \ \rightarrow \ B[x := \beta \ \Gamma] \end{array}$$

Similarly if $\alpha : \Gamma \rightarrow \mathsf{1}$ we can solve it with

$$\alpha \ := \ \lambda \Gamma . \, \langle \rangle$$

This means that any arguments of type $\mathsf{1}$ can safely be left implicit, since they can always be inferred. This might not sound very useful, but in the presence of computed types it becomes interesting. Consider the following safe division function where the proof that the denominator is non-zero is left implicit.

$$\mathit{div} \ : \ (n \ m \ : \ \mathit{Nat})\{p \ : \ \mathit{NonZero} \ m\} \ \rightarrow \ \mathit{Nat}$$

Now if we define

$$\begin{array}{lll} \mathit{NonZero} \ : \ \mathit{Nat} & \rightarrow & \mathsf{Set} \\ \mathit{NonZero} \ \mathsf{zero} & = & 0 \\ \mathit{NonZero} \ (\mathsf{suc} \ n) & = & 1 \end{array}$$

where $\mathsf{0}$ is the empty type. The proof that any number starting with suc is non-zero will be inferred automatically. For instance, we have

$$\mathit{div} \ n \ (\mathsf{suc} \ (\mathsf{suc} \ \mathsf{zero})) \ : \ \mathit{Nat}$$

where the proof that two is non-zero has been instantiated automatically. In the case that $\mathit{NonZero} \ m$ does not reduce to $\mathsf{1}$ the proof will have to be given explicitly.

3.7.2 Function types as terms

Allowing function types as terms poses a bigger challenge. This means that metavariables can be instantiated with function types, and so every time we expect a function type we have to consider the possibility that we encounter a metavariable. This happens when type checking a λ and inferring the type of an application. In these cases we know that the type has to be a function type, so it is safe to instantiate the metavariable thusly. In case the metavariable is not applied to distinct variables the type checking problem has to be postponed, awaiting an instantiation of the metavariable. This means that we have to extend the signature with constants that are waiting to be type checked.

We also have to take into account that metavariable types might appear when conversion checking terms. In this case conversion checking has to be postponed, since we do not know what η-rules to apply.

3.7.3 Universe hierarchy

In the presence of a universe hierarchy the logic has to be extended by level metavariables. This is because when we instantiate a metavariable with a function type as described above, we do not know what levels the new metavariables should live at. It is unclear at this point how to handle the interaction between universe subtyping and level metavariables, since this will introduce inequality constraints between the variables, rather than equality constraints. The current implementation turns unsolved inequalities into equality constraints, which will necessarily exclude valid solutions. The alternative of keeping the inequality constraints and attempting to solve them globally after type checking is potentially very costly.

3.7.4 Pattern matching

If we have definitions by pattern matching, reduction to weak head normal form might be blocked by an uninstantiated metavariable. For instance $\neg\ \alpha$ cannot be reduced to weak head normal for

$$
\begin{aligned}
&\neg\ :\ \mathit{Bool}\ \rightarrow\ \mathit{Bool}\\
&\neg\ \mathsf{true}\ =\ \mathsf{false}\\
&\neg\ \mathsf{false}\ =\ \mathsf{true}
\end{aligned}
$$

Since conversion checking is done on weak head normal forms we generate a constraint when encountering a blocked term.

3.8 Summary

In this chapter we have described an algorithm for type checking for a dependently typed logic extended with metavariables based on McBride's implementation of metavariables in Epigram [McB07]. To maintain the important invariant that terms being evaluated are type correct we work with well-typed approximations of terms, where potentially ill-typed subterms have been replaced by constants. We showed that type checking is decidable and that the algorithm is sound.

We present the type checking algorithm for a simple dependently typed logical framework **MLF**, but it can be extended to more advanced logics. This is evidenced by the fact that we have implemented the algorithm for the Agda language, supporting for instance, definitions by pattern matching, a hierarchy of universes and constants with variable arity. The algorithm has proven to work well with examples of several thousand metavariables.

Chapter 4

Module System

An important feature in any programming language is the possibility of structuring large programs into separate units or modules, and limit the interaction between these modules. This chapter presents a simple, but powerful module system for a dependently typed language. The main focus of the module system is to manage the name space of a program—separating scope checking (name manipulation) from type checking (data manipulation). Data manipulation such as the modeling of algebraic structures is instead done with record types. This separation between the module system and the type checker makes the module system largely independent of the underlying language.

4.1 Introduction

The module system described here has drawn inspiration from a number of sources. First, and perhaps foremost, from the module system of Haskell [PHe+99] which has a similar view on modules as rather passive entities that does not move around much. However, the module system presented here supports many more features than the Haskell module system, such as nested and parameterised modules. Another source of inspiration was Cayenne [Aug98], whose module system also aspired to separate itself from the type checker, but whereas Cayenne modules were split into an interface (type) and an implementation (value), we choose to let modules be defined solely by their implementations.

Another noteworthy module system is the module system of Coq [Chr03] which is based on the ML module system [MTH90]. Similarly to Cayenne, Coq modules are split between interfaces and implementations, and in Coq this is taken one step further allowing, for instance, higher order modules

$$
\begin{array}{rcl}
\mathit{decl} & ::= & [\,\textbf{private}\,]\ \textbf{module}\ M\ \Delta\ \textbf{where}\ \mathit{decls} \\
& | & [\,\textbf{private}\,]\ \textbf{module}\ M_1\ \Delta\ =\ M_2\ \mathit{terms}\ \mathit{mods} \\
& | & \textbf{open}\ M\ [\,\textbf{public}\,]\ \mathit{mods} \\
& | & \textbf{import}\ M_1\ [\,\textbf{as}\ M_2\,]\ \mathit{mods} \\
& | & [\,\textbf{private}\,]\ \mathit{defn} \\
\mathit{mod} & ::= & \textbf{using}\ (\mathit{atom};\ \ldots) \quad | \quad \textbf{hiding}\ (\mathit{atom};\ \ldots) \\
& | & \textbf{renaming}\ (\mathit{atom}\ \textbf{to}\ \mathit{name};\ \ldots) \\
\mathit{atom} & ::= & \mathit{name} \quad | \quad \textbf{module}\ \mathit{name}
\end{array}
$$

Figure 4.1: Module system syntax

(functors) mapping modules implementing a particular interface to a module implementing a different interface. Although the module system of Coq is much more powerful than the module system presented here, it is also significantly more complex.

Harper and Pfenning [HP98] presents a module system for LF in the same spirit as the module system of Coq, and Courant [Cou07] gives a theoretical foundation for this kind of module systems in the context of Pure Type Systems [Bar92a].

While we are trying our best to decouple modules from record types, Pollack takes the opposite approach [Pol00, CPT] and extends record types with more module system like features, such as manifest fields.

4.2 Description

The syntax of the module system is given in Figure 4.1. We leave the syntax of definitions open, since it is not important for the module system. The examples use some suitable made-up syntax, or in the case of the example in Section 4.4, Agda syntax (see Chapter 5).

First let us introduce some terminology. A *definition* is a syntactic construction defining an entity such as a function or a datatype. A *name* is a string used to identify definitions. The same definition can have many names and at different points in the program it will have different names. It may also be the case that two definitions have the same name. In this case there will be an error if the name is used.

The main purpose of the module system is to structure the way names are used in a program. This is done by organising the program in an hierarchical structure of modules where each module contains a number of definitions and

submodules. For instance,

```
module Main where
  Nat : Set
  module B where
    f : Nat → Nat
  g : Nat → Nat → Nat
```

Note that we use indentation to indicate which definitions are part of a module. In the example f is in the module $Main.B$ and g is in $Main$. How to refer to a particular definition is determined by where it is located in the module hierarchy. Definitions from an enclosing module are referred to by their given names as seen in the type of f above. To access a definition from outside its defining module a qualified name has to be used.

```
module Main where
  Nat : Set
  module B where f : Nat → Nat
  ff x = B.f (B.f x)
```

To be able to use the short names for definitions in a module the module has to be *opened*.

```
module Main where
  Nat : Set
  module B where f : Nat → Nat
  open B
  ff x = f (f x)
```

If $A.qname$ refers to a definition d then after **open** A, $qname$ will also refer to d. Note that $qname$ can itself be a qualified name. Opening a module only introduces new names for a definition, it never removes the old names. The policy is to allow the introduction of ambiguous names, but give an error if an ambiguous name is used.

4.2.1 Private definitions

To make a definition inaccessible outside its defining module it can be declared **private**. A private definition is treated as a normal definition inside the module that defines it, but outside the module the definition has no name. In a dependently type setting there are some problems with private definitions—since the type checker performs computations, private names

might show up in goals and error messages. Consider the following (contrived) example

```
module Main where
  module A where
    private IsZero' : Nat → Set
            IsZero' zero    = ⊤
            IsZero' (suc n) = ⊥
    IsZero : Nat → Set
    IsZero n = IsZero' n
  open A
  prf : (n : Nat) → IsZero n
  prf n = ?0
```

The type of the goal $?_0$ is *IsZero n* which normalises to *IsZero′ n*. The question is how to display this normal form to the user. At the point of $?_0$ there is no name for *IsZero′*. One option could be try to fold the term and print *IsZero n*. This is a very hard problem in general, so rather than trying to do this we make it clear to the user that *IsZero′* is something that is not in scope and print the goal as *.Main.A.IsZero′ n*. The leading dot indicates that the entity is not in scope. The same technique is used for definitions that only have ambiguous names.

In effect using private definitions means that from the user's perspective we do not have subject reduction. This is just an illusion, however—the type checker has full access to all definitions.

4.2.2 Name modifiers

An alternative to making definitions private is to exert finer control over what names are introduced when opening a module. This is done by qualifying an open statement with one or more of the modifiers **using**, **hiding**, or **renaming**. You can combine both **using** and **hiding** with **renaming**, but not with each other. The effect of

open A **using** $(\bar{x})$ **renaming** $(\bar{y}$ **to** $\bar{z})$

is to introduce the names $\bar{x}$ and $\bar{z}$ where x_i refers to the same definition as $A.x_i$ and z_i refers to $A.y_i$. Note that if $\bar{x}$ and $\bar{y}$ overlap there will be two names introduced for the same definition. We do not permit $\bar{x}$ and $\bar{z}$ to overlap. The other forms of opening are defined in terms of this one. Let A denote all the (public) names in A. Then

open A **renaming** $(\bar{y}$ **to** $\bar{z})$

$\equiv$ **open** A **hiding** () **renaming** ($\bar{y}$ **to** $\bar{z}$)

open A **hiding** ($\bar{x}$) **renaming** ($\bar{y}$ **to** $\bar{z}$)
$\equiv$ **open** A **using** (A $-$ $\bar{x}$ $-$ $\bar{y}$) **renaming** ($\bar{y}$ **to** $\bar{z}$)

An omitted **renaming** modifier is equivalent to an empty renaming.

4.2.3 Re-exporting names

A useful feature is the ability to re-export names from another module. For instance, one may want to create a module to collect the definitions from several other modules. This is achieved by qualifying the open statement with the **public** keyword:

```
module Nat where
  Nat : Set
module Bool where
  Bool : Set
module Prelude where
  open Nat public
  open Bool public
  isZero : Nat → Bool
```

This will introduce definitions *Prelude.Nat* and *Prelude.Bool* which are visible from outside the *Prelude* module.

4.2.4 Parameterised modules

So far, the module system features discussed have dealt solely with scope manipulation. We now turn our attention to some more advanced features.

It is sometimes useful to be able to work temporarily in a given signature. For instance, when defining functions for sorting lists it is convenient to assume a set of list elements A and an ordering over A. In Coq [BC04] this can be done in two ways: using a *functor*, which is essentially a function between modules, or using a *section*. A section allows you to abstract some arguments from several definitions at once. We introduce parameterised modules analogous to sections in Coq. When declaring a module you can give a telescope of module parameters which are abstracted from all the definitions in the module[1] . For instance, a simple implementation of a sorting function looks like this:

[1]Now we assume some basic properties of our definitions, namely that we can abstract from them. For function definitions this means adding extra arguments, and for datatypes adding more parameters.

```
module Sort (A : Set)(_≤_ : A → A → Bool) where
  insert : A → List A → List A
  insert x ε           = x :: ε
  insert x (y :: ys) with x ≤ y
  insert x (y :: ys) | true  = x :: y :: ys
  insert x (y :: ys) | false = y :: insert x ys

  sort : List A → List A
  sort ε         = ε
  sort (x :: xs) = insert x (sort xs)
```

As mentioned parametrising a module has the effect of abstracting the parameters over the definitions in the module, so outside the *Sort* module we have

```
Sort.insert : (A : Set)(_≤_ : A → A → Bool) →
              A → List A → List A
Sort.sort   : (A : Set)(_≤_ : A → A → Bool) →
              List A → List A
```

For function definitions, explicit module parameter become explicit arguments to the abstracted function, and implicit parameters become implicit arguments. For constructors, however, the parameters are always implicit arguments. This is a consequence of the fact that module parameters are turned into datatype parameters, and the datatype parameters are implicit arguments to the constructors. It also happens to be the reasonable thing to do.

Something which you cannot do in Coq is to apply a section to its arguments. We allow this through the module application statement. In our example:

```
module SortNat = Sort Nat leqNat
```

This will define a new module *SortNat* as follows

```
module SortNat where
  insert : Nat → List Nat → List Nat
  insert = Sort.insert Nat leqNat

  sort : List Nat → List Nat
  sort = Sort.sort Nat leqNat
```

The new module can also be parameterised, and you can use name modifiers to control what definitions from the original module are applied and

what names they have in the new module. The general form of a module application is

module $M_1\ \Delta\ =\ M_2$ *terms modifiers*

A common pattern is to apply a module to its arguments and then open the resulting module. To simplify this we introduce the short-hand

open module $M_1\ \Delta\ =\ M_2$ *terms* [**public**] *mods*

for

module $M_1\ \Delta\ =\ M_2$ *terms mods*
open M_1 [**public**]

We claim that this form of parameterised modules can in many cases replace more advanced module system features such as first class modules and functors. In Section 4.4 we give an example to back up this claim.

4.2.5 Splitting a program over multiple files

When building large programs it is crucial to be able to split the program over multiple files and to not have to type check and compile all the files for every change. The module system offers a structured way to do this. We define a program to be a collection of modules, each module being defined in a separate file. To gain access to a module defined in a different file you can **import** the module:

import M

In order to implement this we must be able to find the file in which a module is defined. To do this we require that the top-level module $A.B.C$ is defined in the file `C.agda` in the directory `A/B/`. One could imagine instead to give a file name to the import statement, but this would mean cluttering the program with details about the file system which is not very nice.

When importing a module M the module and its contents is brought into scope as if the module had been defined in the current file. In order to get access to the unqualified names of the module contents it has to be opened. Similarly to module application we introduce the short-hand

open import M

for

import M
open M

Sometimes the name of an imported module clashes with a local module. In this case it is possible to import the module under a different name.

import M **as** M'

It is also possible to attach modifiers to import statements, limiting or changing what names are visible from inside the module.

4.3 Equipment for record types

A record is essentially a nested Σ-type[2], but in order to use them conveniently we need some basic tools. Two things that one might want are suitably named projection functions and some way of opening a record to bring the fields into scope. It turns out that using the module system we can get both things for the price of one. For a record type

$$
\begin{array}{l}
\textbf{record}\ R\ \Delta\ :\ \mathsf{Set}\ \textbf{where} \\
\quad x_1\ :\ A_1 \\
\quad x_2\ :\ A_2[x_1] \\
\quad \vdots \\
\quad x_n\ :\ A_n[x_1 \ldots\ x_{n-1}]
\end{array}
$$

we generate a parameterised module R

$$
\begin{array}{l}
\textbf{module}\ R\ \{\Delta\}(r\ :\ R\ \Delta)\ \textbf{where} \\
\quad x_1\ :\ A_1 \\
\quad x_1\ =\ \pi_1\ r \\
\quad x_2\ :\ A_2[x_1] \\
\quad x_2\ =\ \pi_1\ (\pi_2\ r) \\
\quad \vdots \\
\quad x_n\ :\ A_n[x_1 \ldots\ x_{n-1}] \\
\quad x_n\ =\ \pi_2\ (\ldots\ (\pi_2\ r))
\end{array}
$$

The functions in R are exactly the projection functions for the record type R. For instance, we have $R.x_2\ :\ \{\Delta\}(r\ :\ R\ \Delta)\ \rightarrow\ A_2[R.x_1\ r]$[3]. Here it is clear that we want the parameters to the record to be implicit regardless of

[2] But with name equality.

[3] So, what in some languages is written $r.x_2$ for $r : R$, we write as $R.x_2\ r$.

whether they are implicit arguments to the record or not. Now the nice thing is that we can apply the module to a given record, effectively projecting all fields from the record at once, which is exactly what we are looking for in a record open feature. So to open a record $r : R$ we simply say

$$\textbf{open module}\ M\ =\ R\ r$$

In the next section we give a bigger example of how to use the module system and the record types together. From now on we will be a bit sloppy with the distinction between a record type and a record (i.e. an element of a record type), and use record for both when it is clear from the context which interpretation is the intended one.

4.4 An example

As an example we will develop some simple lattice theory. We start by defining a record for partial orders. We assume definitions for basic propositional logic.

$$\begin{array}{l}
\textbf{record}\ \mathit{PartialOrder}\ (A\ :\ \mathsf{Set})\ :\ \mathsf{Set}_1\ \textbf{where} \\
\quad _{=}{=}_ \quad\ :\ A \to A \to \mathsf{Set} \\
\quad _{\leqslant}_ \quad\ :\ A \to A \to \mathsf{Set} \\
\quad {=}{=}{-}\mathit{def}\ :\ \{x\ y\ :\ A\} \to (x == y) \iff (x \leqslant y) \wedge (y \leqslant x) \\
\quad {\leqslant}{-}\mathit{refl}\ :\ \{x\ :\ A\} \to x \leqslant x \\
\quad {\leqslant}{-}\mathit{trans}\ :\ \{x\ y\ z\ :\ A\} \to x \leqslant y \to y \leqslant z \to x \leqslant z
\end{array}$$

The reason for including the equality in the record and requiring a proof that it is compatible with the ordering is that most sets already have an equality defined and this way all the theorems about partial orders will use this equality rather than the awkward equality induced by the ordering.

We get a module with projection functions for *PartialOrder* but we would like to also include some derived functions to the projections so we define a new module:

$$\textbf{module}\ \mathit{PartialOrderOps}\ \{A\ :\ \mathsf{Set}\}(\mathit{po}\ :\ \mathit{PartialOrder}\ A)\ \textbf{where}$$

```
  — We want the projection functions to be part of this module
private open module PO = PartialOrder po public
  — We can define some derived functions
_⩾_ : A → A → Set
x ⩾ y = y ⩽ x
  — and prove some auxiliary lemmas. Proofs omitted.
⩽−antisym : {x y : A}   → x ⩽ y → x ⩾ y → x == y
==−refl   : {x : A}     → x == x
==−sym    : {x y : A}   → x == y → y == x
==−trans  : {x y z : A} → x == y → y == z → x == z
  — We also define the dual partial order
dualOrder : PartialOrder A
dualOrder = record { _==_ = _==_
                   ; _⩽_  = _⩾_
                   ; ...
                   }
```

A common idiom when re-exporting the contents of a module M applied to some arguments $\bar{t}$ is

```
private open module M′ = M t̄ public
```

which is equivalent to

```
private module M′ = M t̄
open M′ public
```

That is, we declare a private module M' as the application of M to $\bar{t}$ and then we export the contents of this module. It makes sense to make the intermediate module private, since we export its contents from the current module.

Given a partial order over A and an operation $\_\sqcap\_ : A \to A \to A$ we can define what it means for this to be a semilattice. Since you cannot, in the current presentation, apply or open modules inside the declaration of a record type we put the declaration in a parameterised module. This allows us to apply the *PartialOrderOps* module to our partial order *po* and thus write $x \leqslant y$ rather than *PartialOrderOps*.$\_\leqslant\_$ *po x y*.

```
private
```

```
module IsSemiLatticeDef
    {A : Set}(po : PartialOrder A)(_⊓_ : A → A → A)
    where
  private open module PO = PartialOrderOps po
  record IsSemiLattice : Set where
    ⊓-lbL : {x y : A} → (x ⊓ y) ≤ x
    ⊓-lbR : {x y : A} → (x ⊓ y) ≤ y
    ⊓-glb : {x y z : A} → z ≤ x → z ≤ y → z ≤ (x ⊓ y)
open IsSemiLatticeDef public
```

Now a *SemiLattice* just packs up the partial order and the meet operation with the proofs that they form a semilattice.

```
record SemiLattice (A : Set) : Set₁ where
  po  : PartialOrder A
  _⊓_ : A → A → A
  prf : IsSemiLattice po _⊓_
```

When defining record types like these an interesting question is what should be a parameter to the record and what should be a field in the record. For instance, in this case we could have made the carrier set A be part of the record instead of a parameter. We could also have skipped the *IsSemiLattice* record and put the laws directly in the *SemiLattice* record. Something to keep in mind when faced with this kind of decision is that it is easier to turn a parameter into a field than the other way around. This is exactly what we did with *IsSemiLattice* in the *SemiLattice* record. The reverse operation can be handled by Pollack's manifest fields [Pol00].

Just as for the partial orders we want to extend the projection module with some derived operations. We also include the partial order operations in this module.

```
module SemiLatticeOps {A : Set}(L : SemiLattice A) where
  private
    open module SL   = SemiLattice L          public
    open module SLPO = PartialOrderOps po     public
    open module IsSL = IsSemiLattice po _⊓_ prf public
  ⊓-commute : {x y : A}   → (x ⊓ y) == (y ⊓ x)
  ⊓-assoc   : {x y z : A} → (x ⊓ (y ⊓ z)) == ((x ⊓ y) ⊓ z)
  ⊓-idem    : {x : A}     → (x ⊓ x) == x
```

The real benefit from the module system comes when we define what it means to be a lattice over a set A:

```
open SemiLatticeOps using (dualOrder)
record Lattice (A : Set) : Set₁ where
  sl    : SemiLattice A
  _⊔_   : A → A → A
  prf   : IsSemiLattice (dualOrder sl) _⊔_
```

A lattice over A is a semilattice over A together with a join operation which forms a semilattice with the dual partial order. To get the laws for join we can simply rename the semilattice laws:

```
module LatticeOps {A : Set}(L : Lattice A) where
  private
    module LL = Lattice L
    open module SLL = SemiLatticeOps LL.sl public
         hiding (dualOrder)

    sl' : SemiLattice A
    sl' = record { po   = dualOrder LL.sl; }
                 _⊓_ = LL._⊔_;
                 prf = LL.prf
    open module SLL' = SemiLatticeOps sl' public
      using    ()
      renaming ( ⩽-refl      to ⩾-refl
               ; ⩽-trans     to ⩾-trans
               ; ⩽-antisym   to ⩾-antisym
               ; _⊓_         to _⊔_
               ; ⊓-lbL       to ⊔-ubL
               ; ⊓-lbR       to ⊔-ubR
               ; ⊓-glb       to ⊔-lub
               ; ⊓-commute   to ⊔-commute
               ; ⊓-assoc     to ⊔-assoc
               ; ⊓-idem      to ⊔-idem
               )
    dualLattice : Lattice A
    dualLattice = record { sl = sl'; _⊔_ = _⊓_
                         ; prf = SemiLattice.prf LL.sl }
```

We can play the same trick we did with the dual partial order for lattices. For instance if we prove the left absorption law $x \sqcap (x \sqcup y) == x$ we get the dual law $x \sqcup (x \sqcap y) == x$ simply by instantiating the absorption law to the dual lattice.

This example shows how we can use parameterised modules to exploit symmetries in a program, in this case giving us the join operation and its associated laws for free from the definition of the meet operation.

4.4.1 A note on record subtyping

When using records to model algebraic structures it is sometimes desirable to have a subtyping relation on record types. For instance, a field is a ring with some additional properties, so it would be natural to be able to use a field anytime a ring is required. One way to achieve this is to have the type checker insert coercion functions between fields and rings whenever necessary, which is the way this is done in Coq. A problem with this approach is that it interacts with metavariables in a non-trivial way. Consider the following (contrived) example:

$$
\begin{array}{l}
\textbf{record}\ \mathit{Plus}\ (A\ :\ \mathsf{Set})\ :\ \mathsf{Set}\ \textbf{where} \\
\quad \mathit{plus}\ :\ A\ \to\ A\ \to\ A \\
\textbf{record}\ \mathit{Zero}\ (A\ :\ \mathsf{Set})\ :\ \mathsf{Set}\ \textbf{where} \\
\quad \mathit{zero}\ \ \ \ :\ A \\
\quad \mathit{hasPlus}\ :\ \mathit{Plus}\ A
\end{array}
$$

where *Zero A* is a subtype of *Plus A*. Now assume that $x : \alpha$ where α is a metavariable, and consider the term

$$
\begin{array}{l}
z\ :\ \mathit{Nat} \\
z\ =\ \mathit{Plus.plus}\ x\ (\mathit{Zero.zero}\ x)\ (\mathit{Zero.zero}\ x)
\end{array}
$$

From the first occurrence of x we get the constraint x : *Plus Nat*. Normally, this would allow us to instantiate α with *Plus Nat*, but in this case this would not be correct since the other uses of x require x : *Zero Nat*. One way around this problem would be to introduce some form of row polymorphism [Ré93], but that would require structure equality on record types rather than name equality. Another solution could be to defer instantiation of metavariables until all constraints are known. This is potentially very inefficient.

4.5 Implementation

We now give the details of the scope checking algorithm translating the language in Figure 4.1 to the following much simpler language which can then be type checked.

$$\mathit{decl} ::= \textbf{section}\ M\ \Delta\ \textbf{where}\ \mathit{decls}$$

$$\mid \quad \textbf{apply}\ M_1\ \Delta\ =\ M_2\ \mathit{terms}$$
$$\mid \quad \mathit{defn}$$

As before we leave the syntax of definitions abstract. The only parts of the module system which remain are the parameterised modules and the module applications. These could also be translated away by performing the corresponding abstractions and applications syntactically. This would however mean that the abstracted telescopes, the module arguments, and the types of the definition would be type checked once for each module application and definition in the applied module, so for performance reasons we choose to leave them for the type checker.

4.5.1 Scope checking state

The scope checking algorithm is presented in a monadic style, working on a state consisting of a stack of *scopes*, where each scope corresponds to a module enclosing the declarations being scope checked. A scope has a name which is the name of the corresponding module and a private and a public *name space*. A name space maps names of definitions and modules to unique fully qualified names. We distinguish between the names the user has for definitions and modules (*UDefName*, *UModuleName*) and the unique qualified names used internally by the implementation (*DefName*, *ModuleName*). We use x and y for *UDefName*s, M for *UModuleName*s, z for when both $\mathit{UName} = \mathit{UDefName} \cup \mathit{UModuleName}$, q for *DefName*s, and Q for *ModuleName*s. For the union of $\mathit{Name} = \mathit{DefName} \cup \mathit{ModuleName}$ we use w.

We define

$$\begin{array}{lcll} S & ::= & \varepsilon \mid S \blacktriangleleft \sigma & \text{scope stack} \\ \sigma & ::= & \langle M,\ ns_{pub},\ ns_{pri} \rangle & \text{scopes} \\ ns & ::= & \langle \rho_x,\ \rho_M \rangle & \text{name spaces} \\ \rho_x & \in & \mathit{UDefName} \rightarrow \mathit{Set\ DefName} & \\ \rho_M & \in & \mathit{UModuleName} \rightarrow \mathit{Set\ ModuleName} & \end{array}$$

The same name might at some point refer to several different definitions so a name space maps names to sets of unique names.

The name of a scope is not fully qualified so to get the fully qualified name of an entity z defined in the state S we define FullName z S by

$$\begin{array}{l} \textsc{FullName} : \mathit{UName} \rightarrow \mathit{State} \rightarrow \mathit{Name} \\ \textsc{FullName}\ z\ (\varepsilon \blacktriangleleft \langle M_1,\ \_,\ \_ \rangle \blacktriangleleft \ldots \blacktriangleleft \langle M_n,\ \_,\ \_ \rangle) = M_1.\ldots.M_n.z \end{array}$$

4.5.2 Looking up and adding names

To look up the set of names corresponding to a *UDefName* or *UModuleName* in the state or in a name space we define

$$
\begin{aligned}
&-(-) : \mathit{State} \to \mathit{UName} \to \mathit{Set\ Name} \\
&S(z) = (\textsc{Smash}\ S)(z) \\
&-(-) : \mathit{NameSpace} \to \mathit{UName} \to \mathit{Set\ Name} \\
&\langle \rho_x,\ \rho_M \rangle(x) = \rho_x(x) \\
&\langle \rho_x,\ \rho_M \rangle(M) = \rho_M(M)
\end{aligned}
$$

where $\textsc{Smash}\ S$ takes the union of all name spaces in S.

$$
\begin{aligned}
&\textsc{Smash} : \mathit{State} \to \mathit{NameSpace} \\
&\textsc{Smash}\ S = \bigcup_{\langle \_,\ ns_{pub},\ ns_{pri} \rangle \in S} ns_{pub} \cup ns_{pri}
\end{aligned}
$$

We also define a version which is more suited for a monadic presentation.

$$
\begin{aligned}
&\textsc{Lookup} : \mathit{UName} \to \mathit{State} \to \mathit{Set\ Name} \\
&\textsc{Lookup}\ z\ S = S(z)
\end{aligned}
$$

To bind a name z to a fully qualified name w we simply add the name to the appropriate name space of the top scope. Let $\alpha \in \{pub, pri\}$ indicate whether a name is added to the public or private name space.

$$
\begin{aligned}
&\textsc{bind}_\alpha(- \mapsto -) : \mathit{UName} \to \mathit{Name} \to \mathit{State} \to \mathit{State} \\
&\textsc{bind}_{pub}(z \mapsto w)\,(S \blacktriangleleft \langle M,\ ns_{pub},\ ns_{pri} \rangle) = \\
&\quad S \blacktriangleleft \langle M,\ ns_{pub} \cup \{z \mapsto w\},\ ns_{pri} \rangle \\
&\textsc{bind}_{pri}(z \mapsto w)\,(S \blacktriangleleft \langle M,\ ns_{pub},\ ns_{pri} \rangle) = \\
&\quad S \blacktriangleleft \langle M,\ ns_{pub},\ ns_{pri} \cup \{z \mapsto w\} \rangle
\end{aligned}
$$

4.5.3 Pushing and popping

When entering a module we push an empty scope onto the stack.

$$
\begin{aligned}
&\textsc{push} : \mathit{UModuleName} \to \mathit{State} \to \mathit{State} \\
&\textsc{push}\ M\ S = S \blacktriangleleft \langle M,\ \varepsilon,\ \varepsilon \rangle
\end{aligned}
$$

When popping a scope from the stack, which is done for instance when exiting a module, the public contents of the popped scope becomes available under qualified names. We will want to add the contents to either the public or private name space of the next scope on the stack.

$$
\begin{array}{l}
\textsc{pop}_\alpha \;:\; \mathit{State} \;\to\; \mathit{State} \\
\textsc{pop}_\alpha\,(S \;\blacktriangleleft\; \sigma_2 \;\blacktriangleleft\; \langle M,\; ns_{pub},\; \_\rangle) \;= \\
\quad S \;\blacktriangleleft\; (\textsc{extend}_\alpha\,(\textsc{qualify}_M\; ns_{pub})\,\sigma_2)
\end{array}
$$

In a name space qualified by M all names start with M.

$$
\begin{array}{l}
\textsc{qualify}_\_ \;:\; \mathit{UModuleName} \;\to\; \mathit{NameSpace} \;\to\; \mathit{NameSpace} \\
(\textsc{qualify}_M\; ns)(M.z) \;=\; ns(z) \\
(\textsc{qualify}_M\; ns)(z) \quad\; = \emptyset,\ \textit{if } z \textit{ is not of the form } M.w
\end{array}
$$

To define a name space, we specify how to look up names in it. Since name spaces are essentially functions this constitutes a valid definition.

To add name space ns to a scope σ we write $\textsc{extend}_\alpha\; ns\;\sigma$. Depending on α the name space is added to either the public or private part of σ. To add a scope σ to a scope stack both parts of σ are added to the top scope name space indicated by the argument α.

$$
\begin{array}{l}
\textsc{extend}_\alpha \;:\; \mathit{NameSpace} \;\to\; \mathit{Scope} \;\to\; \mathit{Scope} \\
\textsc{extend}_{pub}\; ns\; \langle M,\; ns_{pub},\; ns_{pri}\rangle \;=\; \langle M,\; ns_{pub} \cup ns,\; ns_{pri}\rangle \\
\textsc{extend}_{pri}\; ns\; \langle M,\; ns_{pub},\; ns_{pri}\rangle \;=\; \langle M,\; ns_{pub},\; ns_{pri} \cup ns\rangle
\end{array}
$$

$$
\begin{array}{l}
\textsc{extend}_\alpha \;:\; \mathit{Scope} \;\to\; \mathit{State} \;\to\; \mathit{State} \\
\textsc{extend}_\alpha\,\langle \_,\; ns_{pub},\; ns_{pri}\rangle\,(S \;\blacktriangleleft\; \sigma) \;= \\
\quad S \;\blacktriangleleft\; \textsc{extend}_\alpha\,(ns_{pub} \;\cup\; ns_{pri})\,\sigma
\end{array}
$$

Remember that the top-level is always a module, so the scope stack will never be empty.

4.5.4 Scope modifiers

We first define the effect of the three scope modifiers on a name space separately. Note that modifying a module name affects all names in that module.

$$
\begin{array}{lcll}
(\textsc{Using}\;\bar{x}\;ns)(z) & = & ns(z), & \textit{if } z \in \bar{x} \\
(\textsc{Using}\;\bar{x}\;ns)(M.z) & = & ns(M.z), & \textit{if } \textbf{module}\, M \in \bar{x} \\
(\textsc{Using}\;\bar{x}\;ns)(z) & = & \emptyset, & \textit{otherwise} \\
(\textsc{Hiding}\;\bar{x}\;ns)(z) & = & ns(z), & \textit{if } z \notin \bar{x} \\
(\textsc{Hiding}\;\bar{x}\;ns)(M.z) & = & ns(M.z), & \textit{if } \textbf{module}\, M \notin \bar{x} \\
(\textsc{Hiding}\;\bar{x}\;ns)(z) & = & \emptyset, & \textit{otherwise} \\
(\textsc{Renaming}\;(\bar{x}\;\textbf{to}\;\bar{y})\;ns)(y_i) & = & ns(x_i) & \\
(\textsc{Renaming}\;(\bar{x}\;\textbf{to}\;\bar{y})\;ns)(y_i.z) & = & ns(M.z), & \textit{if } x_i = \textbf{module}\, M \\
(\textsc{Renaming}\;(\bar{x}\;\textbf{to}\;\bar{y})\;ns)(z) & = & \emptyset, & \textit{otherwise}
\end{array}
$$

Remember (from Section 4.2.2) that we want the source of **renaming**s to be hidden when not explicitly exposed by a **using** clause, so when combining **hiding** and **renaming** we add the renamed names to the hidden ones.

$$\begin{aligned}
&\textsc{ApplyMods}\ (\textbf{using}\ \bar{x}\ \textbf{renaming}\ (\bar{y}\ \textbf{to}\ \bar{z})) \\
&\quad = \textsc{Using}\ (\bar{x}) \cup \textsc{Renaming}\ (\bar{y}\ \textbf{to}\ \bar{z}) \\
&\textsc{ApplyMods}\ (\textbf{hiding}\ \bar{x}\ \textbf{renaming}\ (\bar{y}\ \textbf{to}\ \bar{z})) \\
&\quad = \textsc{Hiding}\ (\bar{x} \cup \bar{y}) \cup \textsc{Renaming}\ (\bar{y}\ \textbf{to}\ \bar{z})
\end{aligned}$$

4.5.5 Scope checking

To make the scope checking algorithm easier to read we make the threading of the scope stack implicit and present the algorithm in a monadic style:

$$\begin{array}{lcl}
x_1 \leftarrow m_1 & & \textbf{let}\ \langle x_1,\ S_1\rangle\ =\ m_1\ S_0 \quad \textbf{in} \\
\ldots & \equiv & \ldots \\
x_n \leftarrow m_n & & \textbf{let}\ \langle x_n,\ S_n\rangle\ =\ m_n\ S_{n-1} \quad \textbf{in} \\
\mathit{return}\ y & & \langle y,\ S_n\rangle
\end{array}$$

where $m_i : \mathit{State} \to A \times \mathit{State}$. We omit the variable binding when we do not care about the result or when there is no result (i.e. $m : \mathit{State} \to \mathit{State}$). For $m : \mathit{State} \to A$ we write $x \leftarrow m$ rather than $x \leftarrow (\lambda S.\ \langle m\ S, S\rangle)$.

The scope stack manages the defined names currently in scope, but we also need to take care of lambda bound names. Thus, scope checking is performed relative to a context Γ of bound names. For A ranging over the sets of things that can be scope checked (such as declarations or terms), and $a \in A$ we define $\Gamma \vdash \textsc{ScopeCheck}(a) : \mathit{State} \to A \times \mathit{State}$.

A module declaration is scope checked as follows.

$$\begin{aligned}
&\Gamma \vdash \textsc{ScopeCheck}(\alpha\ \textbf{module}\ M\ \Delta\ \textbf{where}\ \mathit{decls}) = \\
&\quad \textsc{push}\ M \\
&\quad \Delta' \leftarrow \Gamma \vdash \textsc{ScopeCheck}(\Delta) \\
&\quad \mathit{decls}' \leftarrow \Gamma\Delta' \vdash \textsc{ScopeCheck}(\mathit{decls}) \\
&\quad \textsc{pop}_\alpha \\
&\quad Q \leftarrow \textsc{FullName}\ M \\
&\quad \textsc{bind}_\alpha\ M\ Q \\
&\quad \mathit{return}\ (\textbf{section}\ Q\ \Delta'\ \mathit{decls}')
\end{aligned}$$

If the module was declared **private**, α will be *pri* otherwise it will be *pub*. The declarations in a module are scope checked with an empty scope pushed onto the stack. When we have finished checking the declarations we pop the

scope, which now contains all the names defined in the module, and add it to the next scope on the stack. We also have to bind the name of the defined module. The output is a **section**.

Scope checking a module application is a little more involved. Basically to define a module M_1 as the application of M_2 we open M_2 into a new scope named M_1. However, since module applications introduce new definitions we have to change the qualified names pointing into M_2 so that they point to M_1 instead.

$$
\begin{aligned}
&\Gamma \vdash \textsc{ScopeCheck}(\alpha\ \mathbf{module}\ M_1\ \Delta\ =\ M_2\ \mathit{terms}\ \mathit{mods})\ = \\
&\quad Q_1 \quad\ \leftarrow \textsc{FullName}\ M_1 \\
&\quad Q_2 \quad\ \leftarrow \textsc{Lookup}(M_2) \\
&\quad \Delta' \quad\ \leftarrow \Gamma \vdash \textsc{ScopeCheck}(\Delta) \\
&\quad \mathit{terms}' \leftarrow \Gamma\Delta' \vdash \textsc{ScopeCheck}(\mathit{terms}) \\
&\quad \textsc{push}\ M_1 \\
&\quad \textsc{Open}\ M_2\ \mathit{pub}\ \mathit{mods} \\
&\quad \textsc{Redirect}\ (Q_2\ \mapsto\ Q_1) \\
&\quad \textsc{pop}_\alpha \\
&\quad \textsc{bind}_\alpha\ M_1\ Q_1 \\
&\quad \mathit{return}\ (\mathbf{apply}\ Q_1\ \Delta'\ =\ M_2\ \mathit{terms}')
\end{aligned}
$$

Opening a module M is done by adding all names $M.z$ to the current scope as z, possibly hiding or renaming some names.

$$
\begin{aligned}
&\textsc{Open}\ M\ \alpha\ \mathit{mods}\ (S\ \blacktriangleleft\ \sigma)\ =\ S\ \blacktriangleleft\ \textsc{extend}_\alpha\ \mathit{ns}\ \sigma \\
&\quad \mathbf{where}\ \mathit{ns}\ =\ \textsc{ApplyMods}\ \mathit{mods}\ (\textsc{Match}_M\ (\textsc{Smash}\ (S\ \blacktriangleleft\ \sigma)))
\end{aligned}
$$

where $(\textsc{Match}_M\ \mathit{ns})(x)\ =\ \mathit{ns}(M.x)$.

The redirection of the names from the applied module is defined by

$$
\begin{aligned}
&\textsc{Redirect}\ (Q_2\ \mapsto\ Q_1)\ (S\ \blacktriangleleft\ \sigma)\ =\ \textsc{Redirect}\ (Q_2\ \mapsto\ Q_1)\ \sigma \\
&(\textsc{Redirect}\ (Q_2\ \mapsto\ Q_1)\ \sigma)(z)\quad\ =\ \{\ Q_1.q\ \ \mid\ \ Q_2.q\ \in\ \sigma(z)\ \}
\end{aligned}
$$

For this to be correct it is important that the public names in M_2 all refer to definitions in M_2. That is, we have to make sure that every time we add a name to the public name space of a module it refers to a definition from that module. In particular we have to take care when opening modules publicly.

Ideally we would like to define the scope checking of an **open** statement simply as a call to Open but as just observed this would not be correct in the case of a public open. In this case we create a dummy module which we then open.

$$
\Gamma \vdash \textsc{ScopeCheck}(\mathbf{open}\ M\ \mathit{mods})\ =
$$

$Q \leftarrow$ LOOKUP(M)
OPEN Q *pri mods*
return ε
$\Gamma \vdash$ SCOPECHECK(**open** M **public** *mods*) $=$
$M' \leftarrow$ DUMMYNAME
decls $\leftarrow \Gamma \vdash$ SCOPECHECK(**private module** $M' = M$ *mods*)
$Q \leftarrow$ LOOKUP(M')
OPEN Q *pub*
return decls

The last part of the module system we have to deal with is importing modules from other files. We assume a function FETCHMODULE which finds the file corresponding to a module, scope checks it, and returns the resulting scope.

$\Gamma \vdash$ SCOPECHECK(**import** M_1 **as** M_2 *mods*)
$\sigma \leftarrow$ FETCHMODULE M_1
PUSH M_2
EXTEND$_{pri}$ σ
OPEN M_1 *pub mods*
POP$_{pri}$

4.5.6 Type checking

The only part of the module system left to the type checker is to take care of the parameterised modules and module applications.

To do this we need to keep track of which section we are currently processing as well as the parameters of previously defined sections. We extend the signature with section mappings $Q(\Delta)$ associating the parameters Δ with a section Q. Here Δ contains all parameters to Q including those bound by sections enclosing Q. We also annotate the context with sections. So the context will have the form $M_1(\Delta_1) \ldots M_n(\Delta_n)\Gamma$, where Γ contains the variables bound in a left hand side or in a term. We may combine several sections into one and write this as $Q(\Delta_1 \ldots \Delta_n)\Gamma$, if $Q = M_1 ... M_n$.

For definitions inside a section the type checker should generate definitions with the section parameters abstracted. For instance,

section M $(X$: Set$)$ **where**
$M.id$: $X \rightarrow X$
$M.id\ x = x$

will be translated to

$$M.id \ : \ (X \ : \ \mathsf{Set}) \ \to \ X \ \to \ X$$
$$M.id \ X \ x \ = \ x$$

The judgement form for checking declarations is

$$Q(\Gamma) \ \vdash_\Sigma \ decl \ \rightsquigarrow \ \Sigma'$$

and the rules for modules and definitions are

$$\frac{Q(\Gamma) \vdash_\Sigma \Delta \ \mathbf{ctx} \rightsquigarrow \Delta' \qquad Q(\Gamma)\, M(\Delta') \vdash_\Sigma decls \rightsquigarrow \Sigma'}{Q(\Gamma) \vdash_\Sigma \mathbf{section}\ Q.M\ \Delta\ \mathbf{where}\ decls \ \rightsquigarrow \Sigma', Q.M(\Gamma\Delta')}$$

$$\frac{Q(\Gamma) \vdash_\Sigma e_1 \downarrow \mathsf{Set}_i \rightsquigarrow A \qquad \Sigma' \ = \ \Sigma, Q.f : \Gamma \to A \qquad Q(\Gamma) \vdash_{\Sigma'} e_2 \uparrow A \rightsquigarrow t}{Q(\Gamma) \vdash_\Sigma Q.f : e_1 = e_2 \rightsquigarrow \Sigma, Q.x : \Gamma \to A = \lambda\Gamma.t}$$

To ease the presentation the definition rule is for a simplified form of definition $Q.f \ : \ A = t$. The principle is the same for more advanced forms of definitions, however.

For module applications we generate new definitions applying the definitions from the applied module:

$$\mathbf{apply}\ M' \ = \ M \ Nat$$

turns into

$$M'.id \ : \ Nat \ \to \ Nat$$
$$M'.id \ = \ M.id \ Nat$$

Here we are making the further assumption on the underlying language that it supports definitions of the form $x : A = t$. The rule is

$$\frac{\begin{array}{c} Q_1(\Gamma_1)\, Q_2(\Gamma_2) \vdash_\Sigma \Delta \ \mathbf{ctx} \rightsquigarrow \Delta' \\ Q_1.Q_4(\Gamma_1\Gamma_4) \in \Sigma \qquad Q_1(\Gamma_1)\, Q_2(\Gamma_2)\, \Delta' \vdash_\Sigma \bar{e} : \Gamma_4 \rightsquigarrow \bar{t} \\ \textit{for each } Q_1.Q_4.f_i : \Gamma_1\Gamma_4 \to A_i \in \Sigma \\ \textit{let } \delta_i \ = \ Q_1.Q_2.Q_3.f_i : \Gamma_1\Gamma_2\Delta' \to A_i[\Gamma_4 := \bar{t}] = \lambda\Gamma_1\Gamma_2\Delta'.\ Q_1.Q_4.f_i\ \Gamma_1\ \bar{t} \end{array}}{Q_1(\Gamma_1)\, Q_2(\Gamma_2) \vdash_\Sigma \mathbf{apply}\ Q_1.Q_2.Q_3\ \Delta = Q_1.Q_4\ \bar{e} \rightsquigarrow \Sigma, \bar{\delta}}$$

To better understand what is going on in this rule it helps to look at what the program looks like at the time this rule is applied:

$$\begin{array}{l} \mathbf{module}\ Q_1\ \Gamma_1\ \mathbf{where} \\ \quad \mathbf{module}\ Q_4\ \Gamma_4\ \mathbf{where} \\ \quad\quad Q_1.Q_4.f_i \ : \ A_i \\ \quad \mathbf{module}\ Q_2\ \Gamma_2\ \mathbf{where} \\ \quad\quad \mathbf{apply}\ Q_1.Q_2.Q_3\ \Delta \ = \ Q_1.Q_4\ \bar{e} \end{array}$$

Note that we get one new definition for each definition of the applied module, regardless of whether it was private or hidden when applying the module. These extra definition are unnecessary but harmless and a side effect of our efforts to keep the scope checking and type checking separate.

Inside a parameterised module the parameters have not yet been abstracted over the definitions in the module. In the example below, the type of f depends on from which module it is accessed:

section $A\ (X\ :\ \mathsf{Set})$ **where**
  section $A.B\ (Y\ :\ \mathsf{Set})$ **where**
    $A.B.f\ :\ X\ \to\ Y\ \to\ X$
  — Inside $A.B$ we have $A.B.f : X \to Y \to X$
  — Outside $A.B$ we have $A.B.f : (Y : \mathsf{Set}) \to X \to Y \to X$
  — Outside A we have $A.B.f : (X : \mathsf{Set})(Y : \mathsf{Set}) \to X \to Y \to X$

Since the type checker removes all sections it will have to add the missing arguments to uses of $A.B.f$ inside A and $A.B$. The rule for inferring the type of a defined constant is

$$\frac{Q_1.q : \Delta_1 \to A \in \Sigma}{Q_1(\Delta_1)Q_2(\Delta_2)\Gamma \vdash_\Sigma Q_1.q \downarrow A \rightsquigarrow Q_1.q\ \Delta_1}$$

That is inside the module Q_1 parameterised by Δ_1 functions defined in Q_1 are automatically applied to the parameters Δ_1.

4.6 Summary

We have presented a reasonable simple and easy to implement module system which is still expressive enough to allow large programs to be structured in a nice way. A key design decision was to keep the module system and the type system as separate as possible. The result is that only parameterised modules survive into the type checking phase and they can be handled with relatively small modifications to the type checking algorithm. In short, the module system consists of name space management and λ-lifting [Joh85]. The module system also supports separate compilation, to the extent possible in a dependently typed language. Previously defined modules do not need to be re-type checked, but we do need access to their defined functions in order to perform the necessary computations during type checking.

To demonstrate the module system we gave an example of a library of lattice theory where the module system was used in a crucial way to get dual properties for free.

Chapter 5

The Agda Language

The ideas described in the previous chapters have all been incorporated into a language Agda which is readily available for download on the web [Nor07]. The language is a redesign and reimplementation of the Agda language by Coquand and Coquand [CC99]. The version described in this chapter is 2.1.0, and consists of around 30,000 lines of Haskell code in 150 modules.

Agda has gathered a few users and some quite impressive work has been done using it [AC07, AMS07, BD07, Dan06, Dan07, SA07]. There was also a course on Agda in the TYPES summer school 2007 [ACT07].

Where the previous chapters give the technical details of the features in Agda, this chapter provides a description of the language from a user's perspective.

5.1 Language description

5.1.1 Names

A *name part* is a non-empty sequence of printable Unicode characters not containing any of the following reserved characters.

Reserved characters: `@.(){};_`

Furthermore there is a set of reserved words given in Figure 5.1 that cannot be used as name parts. This means that strings like `x:A` and `A->B` are valid *names*. To write the type signature and the function type, white space have to be inserted: `x : A`, and `A -> B`.

A *name* is a non-empty sequence of alternating name parts and `_` (excluding the singleton `_`). A name containing `_` can be used as an operator where the arguments go in place of the `_`. For instance, an application of

`[0-9]`$^+$ `->` `:` `=` `?` `\` `|` `Prop` `Set`[0-9]* `abstract`
`data` `forall` `hiding` `import` `in` `infix` `infixl`
`infixr` `let` `module` `mutual` `open` `postulate` `primitive`
`private` `public` `record` `renaming` `using` `where` `with`

Figure 5.1: Reserved words

the name `if_then_else_` to arguments x, y, and z can be written either as a normal application `if_then_else_` x y z or as an operator application `if` x `then` y `else` z.

A *qualified name* is a non-empty sequence of names separated by `.` (dot). Qualified names are used to refer to entities in other modules.

5.1.2 Interaction points

Interaction points are holes in a program where an expression should be filled in. These are written `?` or `{!...!}`. In an interactive environment the user can interact with the type checker through these interaction points, for instance, asking for the type of the expression to be filled in or the local context.

Internally the type checker treats interaction points as metavariables which will not be solved automatically.

5.1.3 Implicit syntax

It is possible to omit terms that the type checker can figure out for itself, replacing them by `_`. If the type checker cannot infer the value of an `_` it will report an error. For instance, for the polymorphic identity function $id : (A : \mathsf{Set}) \to A \to A$, the first argument can be inferred from the type of the second argument, so we might write `id _ zero` for the application of the identity function to zero. The implicit syntax is implemented using the metavariables described in Chapter 3.

5.1.4 Functions

Function types are written `(x : A) -> B` or `A -> B` for non-dependent functions. Function types can range over arbitrary telescopes, for instance, the

type of the substitutivity law for a polymorphic equality `Eq : (A : Set) -> A -> A -> Set` can be stated as

```
(A : Set)(C : A -> Set)(x y : A) -> Eq A x y -> C x -> C y
```

Functions are constructed by lambda abstractions, which can be either typed or untyped. For instance, both expressions below have type `(A : Set) -> A -> A` (the second expression checks against other types as well):

```
\ (A : Set)(x : A) -> x
\ A x -> x
```

5.1.5 Implicit arguments

Implicit function spaces are written with curly braces instead of parenthesis. We can restate our polymorphic equality and substitution principle as

```
_==_  : {A : Set} -> A -> A -> Set
subst : {A : Set}(C : A -> Set){x y : A} -> x == y -> C x -> C y
```

Note how the first argument to `_==_` is left implicit. Similarly we may leave out the implicit arguments A, x, and y in an application of `subst`. To give an implicit argument explicitly, enclose in curly braces. The following two expressions are equivalent:

```
subst C eq cx
subst {_} C {_} {_} eq cx
```

Implicit arguments can also be referred to by name, so if we want to give the expression e explicitly for y without giving a value for x we can write

```
subst C {y = e} eq cx
```

When constructing implicit function spaces the implicit argument can be omitted, so both expressions below are valid expressions of type `{A : Set} -> A -> A`:

```
\ {A} x -> x
\ x -> x
```

There are no restrictions on when a function space can be implicit. Internally, explicit and implicit function spaces are treated in the same way. This means that there are no guarantees that implicit arguments will be solved. When there are unsolved implicit arguments the type checker will give an error message indicating which application contains the unsolved arguments.

The reason for this liberal approach to implicit arguments is that limiting the use of implicit argument to the cases where we guarantee that they are solved rules out many useful cases in practice.

See Section 3.6 for the details on how metavariables are inserted for implicit arguments during type checking.

5.1.6 Datatypes and function definitions

Functions can be introduced by giving a type and a definition. For instance, the polymorphic identity function can be defined by

```
id : {A : Set} -> A -> A
id x = x
```

Note that the implicit argument is left out in the left hand side. As in a lambda abstraction it can be given explicitly by enclosing it in curly braces:

```
id : {A : Set} -> A -> A
id {A} x = x
```

Datatypes are introduced by `data` declarations. For instance, the natural numbers can be defined by

```
  data Nat : Set where
    zero : Nat
    suc  : Nat -> Nat
```

To ensure normalisation, inductive occurrences must appear in strictly positive positions. For instance, the following datatype is not allowed:

```
data Bad : Set where
  bad : (Bad -> Bad) -> Bad
```

since there is a negative occurrence of `Bad` in the argument to the constructor.

Functions over elements of a datatype can be defined using pattern matching and structural recursion. The addition function on natural numbers is defined by

```
  _+_ : Nat -> Nat -> Nat
  zero  + m = m
  suc n + m = suc (n + m)
```

The operator form can be used both in left hand sides and right hand sides as seen here.

Datatypes can be parameterised over a telescope of parameters. These are written after the name of the datatype and scope over the constructors.

```
data List (A : Set) : Set where
  []   : List A
  _::_ : A -> List A -> List A
```

This will introduce the constructors

```
[]   : {A : Set} -> List A
_::_ : {A : Set} -> A -> List A -> List A
```

We can also define inductive families of sets [Dyb94]. For instance, the family over natural numbers n of proofs that n is even.

```
data IsEven : Nat -> Set where
  evenZ  : IsEven zero
  evenSS : (n : Nat) -> IsEven n -> IsEven (suc (suc n))
```

Note the difference between the left and the right side of the first `:`. Types appearing on the left are parameters and scope over the constructors. These have to be unchanged in the return types of the constructors. Types appearing on the right are indices which are not in scope in the constructors, and which take on arbitrary values in the constructor return types.

When pattern matching on an element of an inductive family we get information about the index (see Chapter 2 for the details). To distinguish parts of a pattern which are determined by pattern matching (the inaccessible patterns) and the parts which constitutes the actual pattern matching, the inaccessible patterns are prefixed with a dot. In Chapter 2 these were written $\lfloor t \rfloor$. For instance, we can prove that the sum of two even numbers is also even.

```
even+ : (n m : Nat) -> IsEven n -> IsEven m -> IsEven (n + m)
even+ .zero          m  evenZ        em = em
even+ .(suc (suc n)) m (evenSS n en) em =
    evenSS (n + m) (even+ n m en em)
```

The proof is by recursion on the proof that n is even. Pattern matching on this proof will force the value of n and hence the patterns for n are prefixed with a dot to indicate that they are not part of the pattern matching. In this case we can make n and m implicit and write the proof as

```
even+ : {n m : Nat} -> IsEven n -> IsEven m -> IsEven (n + m)
even+  evenZ        em = em
even+ (evenSS n en) em = evenSS _ (even+ en em)
```

5.1.7 Records

Record types are declared in much the same way as datatypes, but instead of giving the types of the constructors you give the types of the record fields. For instance, we can define the type of even numbers as a record type containing a number and a proof that it is even.

```
record Even : Set where
  val : Nat
  prf : IsEven val
```

Note that later fields may refer to earlier field values by name. Record types are compared by name, so this introduces a new type `Even`, different from all other record types. To build an element of a record type you write

```
record { val = suc (suc zero); prf = evenSS _ evenZ }
```

The fields can be given in any order. For each record type a module of the same name is defined, containing projection functions. In the case of `Even` we have

```
Even.val : Even -> Nat
Even.prf : (e : Even) -> IsEven (Even.val e)
```

The module `Even` containing the projection functions is parameterised over the record and so it can be applied and opened (see Section 4.3 for the details). In case the record is parameterised the generated module have the record parameters as implicit parameters. For instance,

```
record Step (A : Set) : Set where
  next : A -> A
```

will introduce a module

```
module Step {A : Set}(s : Step A) where
  next : A -> A
```

5.1.8 Local definitions

Each clause in a function definition can have a block of local declarations. These can be any declarations that can appear on the top-level, including modules, datatype declarations, and recursive functions. For instance, the `reverse` function can be defined using a local recursive function:

```
reverse : {A : Set} -> List A -> List A
reverse {A} xs = rev xs []
  where
    rev : List A -> List A -> List A
    rev []        ys = ys
    rev (x :: xs) ys = rev xs (x :: ys)
```

As seen, the variables bound in the left hand side of the clause are in scope in the local declarations.

A problem with local declarations is that they are just that—local. In the example above we cannot prove any interesting properties about the `reverse` function, since we do not have access to `rev`. For this reason it is often preferable to use a private definition:

```
private
  rev : {A : Set} -> List A -> List A -> List A
  rev []        ys = ys
  rev (x :: xs) ys = rev xs (y :: ys)

reverse : {A : Set} -> List A -> List A
reverse {A} xs = rev xs []
```

This way properties of the helper function can be proven in the module defining it, but it still will not be accessible outside the module.

5.1.9 Module system

The purpose of the module system is to manage the name space of Agda programs. A program is structured in a number of files, each file containing a single top-level module which in turn may contain any number of submodules. To refer to entities defined in another module its name is qualified by the name of the module. For instance, to refer to `Nat` from outside the `Numbers` module you write `Numbers.Nat`:

```
module Example where
  module Numbers where
    data Nat : Set where
      zero : Nat
      suc  : Nat -> Nat
  one : Numbers.Nat
  one = Numbers.suc Numbers.zero
```

Remember that the extent of a module is determined by indentation. To use the names from a module available without qualification, one uses an `open` statement:

```
open Numbers
two : Nat
two = suc one
```

The full description of the module system can be found in Chapter 4, including parameterised modules, and more fine-grained control over `open` statements.

5.1.10 Additional features

In addition to the features described here, Agda has experimental support for mutual induction-recursive definitions [DS06]. Mutual definitions are given inside a `mutual` block:

```
mutual
  even : Nat -> Bool
  even zero    = true
  even (suc n) = odd n

  odd : Nat -> Bool
  odd zero    = false
  odd (suc n) = even n
```

A detailed discussion of mutual inductive-recursive definitions is beyond the scope of this thesis.

5.2 A bigger example

Dependent types not only gives you the possibility to prove properties about programs, you can also write programs to compute proofs. To illustrate this we develop an internal solver for equations in a commutative monoid, such as the natural numbers with addition and zero. The basic idea is to model such equations by a datatype and define a normalisation function for this datatype. To check if an equation holds we can then simply check that both sides reduces to the same normal form. We prove this strategy sound which enables us to use the solver to prove equations in arbitrary commutative monoids.

This section consists of a number of literate Agda files which can be processed both by LaTeX and the Agda type checker.

5.2.1 Logic

We start out by defining some basic logical connectives in a module `Logic`.

```
module Logic where
```

The false proposition is defined as the empty datatype and the true proposition is the record with no fields.

```
data   False : Set where
record True  : Set where

tt : True
tt = record {}
```

Note that the η-equality for records implies that all elements of `True` are equal.

Disjunction and conjunction are simple datatypes. Note that comma (`,`) is a valid name character. Negation is defined in the usual way as implication of falsity.

```
data _∨_ (A B : Set) : Set where
  inl : A -> A ∨ B
  inr : B -> A ∨ B

data _∧_ (A B : Set) : Set where
  _,_ : A -> B -> A ∧ B

¬_ : Set -> Set
¬ A = A -> False
```

The identity type $x \equiv y$ is only inhabited if x and y are definitionally equal, in which case the unique[1] element is `ref`.

```
data _≡_ {A : Set}(x : A) : A -> Set where
  ref : x ≡ x
```

5.2.2 Basic datatypes

We also need a set of basic datatypes, such as booleans, natural numbers and lists. We define these in a module `Basics`.

```
module Basics where

open import Logic
```

[1] This is not without controversy. See Section 1.5.2 and Section 2.2.2 for the details.

The identity function and function composition are always useful so let us define them.

```
id : {A : Set} -> A -> A
id x = x

_∘_ : {A B : Set}{C : B -> Set} ->
      ((x : B) -> C x) -> (g : A -> B)(x : A) -> C (g x)
(f ∘ g) x = f (g x)
```

The given generalisation of the non-dependent composition function is sometimes useful, and enjoys the property that we can still infer the type arguments. We define the booleans with the constructors `false`, and `true`.

```
data Bool : Set where
  false : Bool
  true  : Bool

infix 5 if_then_else_
if_then_else_ : {A : Set} -> Bool -> A -> A -> A
if true  then x else y = x
if false then x else y = y
```

The fixity of the `if_then_else_` dictates whether or not parenthesis are needed for the else branch. We would like to avoid parentheses so we set it to a low value. A high fixity means that the operator binds tightly and a low fixity that it binds loosely. For instance, given

```
infixl 20 _+_
infixl 30 _*_
```

the expression `x + y * z` parses as `x + (y * z)` rather than `(x + y) * z`.

Natural numbers are defined with two constructors `zero` and `suc`. The `BUILTIN` pragmas tells the type checker about our definition of natural numbers and allows them to be represented more efficiently internally. It also lets us use numeric literals to construct natural numbers.

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat

{-# BUILTIN NATURAL Nat  #-}
{-# BUILTIN ZERO    zero #-}
{-# BUILTIN SUC     suc  #-}
```

A very handy type is the family of finite sets. `Fin` n is the n-element set whose elements are `fzero`, `fsuc fzero`, ..., `fsuc`$^{n-1}$ `fzero`.

```
data Fin : Nat -> Set where
  fzero : {n : Nat} -> Fin (suc n)
  fsuc  : {n : Nat} -> Fin n -> Fin (suc n)
```

We are going to need to compare elements of finite sets, so we define a boolean less than or equals operation.

```
_⩽Fin⩽_ : {n : Nat} -> Fin n -> Fin n -> Bool
fzero  ⩽Fin⩽ x      = true
fsuc n ⩽Fin⩽ fzero  = false
fsuc n ⩽Fin⩽ fsuc m = n ⩽Fin⩽ m
```

Lists are defined as one would expect. We make the cons operation `_::_` right associative.

```
data List (A : Set) : Set where
  []   : List A
  _::_ : A -> List A -> List A

infixr 50 _::_
```

If we index lists by length we get vectors:

```
data Vec (A : Set) : Nat -> Set where
  ε   : Vec A zero
  _►_ : {n : Nat} -> A -> Vec A n -> Vec A (suc n)

infixr 50 _►_
```

Vectors and finite sets have an interesting relationship: `Fin` n is the type of positions in `Vec` A n. Hence, we have an isomorphism between `Vec` A n and `Fin` $n \to A$ as witnessed by the functions `_!_` and `tabulate`. Both of these functions will come in handy later on.

```
infixl 80 _!_

_!_ : {A : Set}{n : Nat} -> Vec A n -> Fin n -> A
ε        ! ()
(x ► xs) ! fzero  = x
(x ► xs) ! fsuc i = xs ! i

tabulate : {A : Set}{n : Nat} -> (Fin n -> A) -> Vec A n
tabulate {n = zero } f = ε
tabulate {n = suc n} f = f fzero ► tabulate (f ∘ fsuc)
```

The natural number argument n to `tabulate` can be inferred by the type checker when we use the function, but defining `tabulate` we need to recurse over n. Rather than also binding A explicitly in the left hand side we refer to n by name. The name to be used is taken from the type.

5.2.3 Equivalence relations

Next we define a small library for equivalence relations and give instances for lists and finite sets which are the ones we need for our solver.

```
module Equivalence where

open import Logic
```

We split the definition of what an equivalence relation is into two parts. First we define what it means for a relation to be an equivalence and then we define an equivalence relation to be a relation and a proof that it is an equivalence. The advantage of this approach as opposed to just having a single record is that we can talk about what it means to be an equivalence. This makes defining more refined equivalence relations, such as decidable equivalence relations, easier.

```
record IsEquivalence {A : Set}(_==_ : A -> A -> Set) : Set
 where
  refl  : (x : A)     -> x == x
  sym   : (x y : A)   -> x == y -> y == x
  trans : (x y z : A) -> x == y -> y == z -> x == z

record Equivalence (A : Set) : Set1 where
  _==_    : A -> A -> Set
  isEquiv : IsEquivalence _==_
```

Now the disadvantage of the two stage approach is that the module generated for the equivalence record does not contain projections for the axioms `refl`, `sym`, and `trans`. For this reason we define a new module `EquivalenceOps` which simply re-exports the projection functions from the two records.

```
module EquivalenceOps {A : Set}(Eq : Equivalence A) where
  private open module Eq   = Equivalence Eq        public
  private open module IsEq = IsEquivalence isEquiv public
```

We now define a type of decidable equivalence relations. The definition is the same as the definition of equivalence relation except we have an extra

axiom. Note that record types are compared by name and there is no subtyping between records, so `Equivalence` and `DecidableEquivalence` are two completely separate types.

```
record DecidableEquivalence (A : Set) : Set1 where
  _==_    : A -> A -> Set
  isEquiv : IsEquivalence _==_
  decide  : (x y : A) -> (x == y) ∨ ¬ (x == y)
```

Just as before, we define a new module with projection functions. This module also provides a function to extract an `Equivalence` from a decidable equivalence relation.

```
module DecidableEquivalenceOps
  {A : Set}(DEq : DecidableEquivalence A)
  where
  private module DEq = DecidableEquivalence DEq
  open DEq public using (decide)

  Eq : Equivalence A
  Eq = record { _==_ = DEq._==_; isEquiv = DEq.isEquiv }

  private open module Eq = EquivalenceOps Eq public
```

Next we give some examples of equivalence relations that we will need later on. First of all we prove that the identity type `_≡_` is an equivalence.

```
open import Basics

identityEquivalence : (A : Set) -> Equivalence A
identityEquivalence A = record
  { _==_ = _≡_
  ; isEquiv = record
    { refl  = \x -> ref
    ; sym   = sym
    ; trans = trans
    }
  }
  where
    sym : (x y : A) -> x ≡ y -> y ≡ x
    sym x .x ref = ref

    trans : (x y z : A) -> x ≡ y -> y ≡ z -> x ≡ z
    trans x .x z ref xz = xz
```

In the proofs of symmetry and transitivity we can see the pattern matching on identity proofs in action.

We define a decidable equivalence relation on finite sets by proving that the identity relation is decidable.

```
finDecEquivalence : {n : Nat} -> DecidableEquivalence (Fin n)
finDecEquivalence {n} = record
  { _==_    = _==_
  ; isEquiv = isEquiv
  ; decide  = decide
  }
  where
    open module E {n : Nat} =
      EquivalenceOps (identityEquivalence (Fin n))

    decide : {n : Nat}(i j : Fin n) -> (i == j) ∨ ¬ (i == j)
    decide fzero    fzero    = inl ref
    decide fzero    (fsuc j) = inr dismiss
      where
        dismiss : fzero == fsuc j -> False
        dismiss ()
    decide (fsuc i) fzero    = inr dismiss
      where
        dismiss : fsuc i == fzero -> False
        dismiss ()
    decide (fsuc i) (fsuc j) with decide i j
    decide (fsuc i) (fsuc .i) | inl ref = inl ref
    decide (fsuc i) (fsuc j)  | inr neq = inr (dismiss i j neq)
      where
        dismiss : (i j : Fin _) ->
                  ¬ (i == j) -> ¬ (fsuc i == fsuc j)
        dismiss i .i neq ref = neq ref
```

Note that when we instantiate the `EquivalenceOps` module to the identity relation on finite sets we abstract over the size of the set. This keeps the operations polymorphic in the size, which we need in the proof. To dismiss the off-diagonal cases we use the syntax for pattern matching on caseless types.

Given an equivalence relation on a type A we can define an equivalence relation on lists over A, relating lists of equal length when the elements are pointwise related. The proofs are simple but somewhat tedious.

```
listEquivalence : {A : Set} ->
                  Equivalence A -> Equivalence (List A)
```

```
listEquivalence {A} eqA = record
  { _==_ = _=List=_
  ; isEquiv = record
    { refl  = reflList
    ; sym   = symList
    ; trans = transList
    }
  }
  where
    open module EqA = EquivalenceOps eqA
    _=List=_ : List A -> List A -> Set
    []        =List= []        = True
    []        =List= (y :: ys) = False
    (x :: xs) =List= []        = False
    (x :: xs) =List= (y :: ys) = (x == y) ∧ (xs =List= ys)

    reflList : (xs : List A) -> xs =List= xs
    reflList []        = tt
    reflList (x :: xs) = (refl x , reflList xs)

    symList : (xs ys : List A) -> xs =List= ys -> ys =List= xs
    symList []        []        eq = eq
    symList []        (_ :: _)  ()
    symList (_ :: _)  []        ()
    symList (x :: xs) (y :: ys) (xy , xsys) =
      (sym x y xy , symList xs ys xsys)

    transList : (xs ys zs : List A) ->
                xs =List= ys -> ys =List= zs -> xs =List= zs
    transList []        []        zs        _  eq = eq
    transList []        (_ :: _)  _         () _
    transList (_ :: _)  []        _         () _
    transList (_ :: _)  (_ :: _)  []        _  ()
    transList (x :: xs) (y :: ys) (z :: zs)
              (xy , xsys) (yz , yszs) =
      (trans x y z xy yz , transList xs ys zs xsys yszs)
```

If the equivalence on the elements is decidable then so is the induced list equivalence.

```
listDecEquivalence : {A : Set} -> DecidableEquivalence A ->
                     DecidableEquivalence (List A)
listDecEquivalence {A} deqA = record
  { _==_    = _==_
```

```
  ; isEquiv = isEquiv
  ; decide  = decide
  }
  where
    module DEqA   = DecidableEquivalenceOps deqA
    open module EqList =
      EquivalenceOps (listEquivalence DEqA.Eq)

    decide : (xs ys : List A) -> (xs == ys) ∨ ¬ (xs == ys)
    decide []        []        = inl _
    decide []        (y :: ys) = inr \w -> w
    decide (x :: xs) []        = inr \w -> w
    decide (x :: xs) (y :: ys)
      with DEqA.decide x y | decide xs ys
    decide (x :: xs) (y :: ys)
         | inl xy  | inl xsys = inl (xy , xsys)
    decide (x :: xs) (y :: ys)
         | inr nxy | _        = inr dismiss
      where
        dismiss : (x :: xs) == (y :: ys) -> False
        dismiss (xy , _) = nxy xy
    decide (x :: xs) (y :: ys)
         | _       | inr nxsys = inr dismiss
      where
        dismiss : (x :: xs) == (y :: ys) -> False
        dismiss (_ , xsys) = nxsys xsys
```

In the case where both lists are non-empty we use a `with` clause to pattern match on the results of comparing the heads and the tails.

5.2.4 Chain reasoning

Constructing equivalence proofs using transitivity directly results in very unreadable proofs. Fortunately we can use a little implicit argument and infix operator magic to solve this problem.

We define a module `Chain` parameterised over a reflexive and transitive relation.

```
module Chain
    {A : Set}(_==_ : A -> A -> Set)
    (refl : (x : A) -> x == x)
    (trans : (x y z : A) -> x == y -> y == z -> x == z)
  where
```

This module exports the following three operators:

```
infix  2 chain>_
infixl 2 _===_by_
infix  1 _qed
```

where `chain>_` starts a proof, `_===_by_` performs one step of the proof, and `_qed` concludes. For instance, given

$$\begin{aligned}&\mathit{commute} : (n\ m : \mathit{Nat}) \to n + m == m + n\\&\mathit{pluszero} : (n : \mathit{Nat}) \to n + 0 == n\end{aligned}$$

we can prove that $0 + n == n$ by

$$\begin{aligned}&\mathit{chain}\rangle\ 0 + n\\&\quad === n + 0 \quad \mathit{by}\ \ \mathit{commute}\ 0\ n\\&\quad === n \qquad \mathit{by}\ \ \mathit{pluszero}\ n\\&\mathit{qed}\end{aligned}$$

Compare this to the same proof using *trans*:

$$\mathit{trans}\ (0 + n)\ (n + 0)\ n\ (\mathit{commute}\ 0\ n)\ (\mathit{pluszero}\ n)$$

which is a lot less readable even though in this case we only needed a single appeal to transitivity.

To make sure that the implicit arguments can be solved regardless of the definition of `_==_` we create a wrapper datatype `_≃_`. There will be no need to refer to this type from the outside so we make it private.

```
private
  data _≃_ (x y : A) : Set where
    prf : x == y -> x ≃ y
```

Now `chain>` is simply reflexivity for the wrapper datatype, and `_==_by_` is transitivity with carefully chosen implicit arguments.

```
chain>_ : (x : A) -> x ≃ x
chain> x = prf (refl x)

_===_by_ : {x y : A} -> x ≃ y -> (z : A) -> y == z -> x ≃ z
prf p === z by q = prf (trans _ _ _ p q)
```

The `_qed` function simply unwraps the constructed proof.

```
_qed : {x y : A} -> x ≃ y -> x == y
prf p qed = p
```

5.2.5 Monoids

So far we have mostly been developing general libraries with no apparent connection to the problem we are trying to solve—that of automatically proving equations in a commutative monoid. We start the problem specific part by defining what a commutative monoid is. This is done relative to a set A equipped with an equivalence relation.

```
open import Equivalence
module Monoid {A : Set}(Eq : Equivalence A) where
```

We want to have access to the operations on equivalence relations so we apply and open the `EquivalenceOps` module.

```
private open module Eq = EquivalenceOps Eq
```

We use the same two stage approach as we did for equivalence relations and first define what it means for an element $\emptyset$ and an operation `_+_` to form a monoid. The definition of a monoid is then simply a $\emptyset$, a `_+_`, and a proof that they form a monoid.

```
record IsMonoid (∅ : A)(_+_ : A -> A -> A) : Set where
  idL   : (x : A)     -> (∅ + x) == x
  idR   : (x : A)     -> (x + ∅) == x
  assoc : (x y z : A) -> (x + (y + z)) == ((x + y) + z)
  cong  : (x₁ x₂ y₁ y₂ : A) ->
          x₁ == x₂ -> y₁ == y₂ -> (x₁ + y₁) == (x₂ + y₂)

record Monoid : Set where
  ∅         : A
  _+_       : A -> A -> A
  isMonoid : IsMonoid ∅ _+_
```

Again we define a new module with the projection functions from both records as well as a couple of derived ones.

```
module MonoidOps (M : Monoid) where
  private open module M   = Monoid M public
  private open module IsM = IsMonoid isMonoid public

  congL : (x y₁ y₂ : A) -> y₁ == y₂ -> (x + y₁) == (x + y₂)
  congL _ _ _ eq = cong _ _ _ _ (refl _) eq

  congR : (x₁ x₂ y : A) -> x₁ == x₂ -> (x₁ + y) == (x₂ + y)
  congR _ _ _ eq = cong _ _ _ _ eq (refl _)
```

Note that the element arguments to `cong` above can all be inferred. This is possible since `_==_` is abstract, but it will not necessarily be the case for concrete values of `_==_`.

A commutative monoid is simply a monoid where addition is commutative. Here we use a different strategy than when we extended equivalence relations to decidable equivalence relations. Instead of repeating the monoid fields we simply add a field which is a monoid. The price we have to pay is that it becomes more cumbersome to refer to the `_+_` operation. One could imagine allowing module application and opening inside record declarations to solve this problem.

```
record CommutativeMonoid : Set where
  monoid  : Monoid
  commute : (x y : A) ->
    MonoidOps._+_ monoid x y == MonoidOps._+_ monoid y x

module CommutativeMonoidOps (M : CommutativeMonoid) where
  private open module C = CommutativeMonoid M public
  private open module M = MonoidOps monoid public
```

In a general monoid library there would of course be a lot more operations and properties, but for our purposes this is enough.

5.2.6 Representing commutative monoid equations

The previous section defined the notion of a commutative monoid but it does not give us any way of analysing expressions in a monoid. In this section we define a datatype of commutative monoid expressions.

```
module Expr where

open import Logic
open import Basics
open import Equivalence
```

The representation of monoid expressions are parameterised by the number of free variables. There are constructors for $\emptyset$ and addition and a constructor for variables. We represent variables by elements in a finite set. The representation of an equation is just a pair of terms.

```
data Expr (n : Nat) : Set where
  |∅|   : Expr n
  _|+|_ : Expr n -> Expr n -> Expr n
  var   : Fin n -> Expr n
```

```
data Equation (n : Nat) : Set where
  _≐_   : Expr n -> Expr n -> Equation n
```

In order to decide whether or not an equation holds we will normalise both sides and compare the normal forms. We chose normal forms to be ordered lists of variables. We do not enforce that the lists are ordered. This is not necessary for soundness, but if we were to prove completeness it might simplify matters.

```
NF : Nat -> Set
NF n = List (Fin n)
```

An alternative, perhaps nicer, representation of normal forms would be as a vector of variable counts: `NF n = Vec Nat n`.

The empty list is the zero of the normal forms and the addition is the merge function of two ordered lists:

```
_⊕_ : {n : Nat} -> NF n -> NF n -> NF n
[]        ⊕ ys        = ys
(x :: xs) ⊕ []        = x :: xs
(x :: xs) ⊕ (y :: ys) = if x ⩽Fin⩽ y
                          then x :: (xs ⊕ (y :: ys))
                          else y :: ((x :: xs) ⊕ ys)
```

To normalise an expression we simply replace `|∅|` with the empty list and `_|+|_` with `_⊕_`. Variables become singleton lists.

```
normalise : {n : Nat} -> Expr n -> NF n
normalise |∅|         = []
normalise (e₁ |+| e₂) = normalise e₁ ⊕ normalise e₂
normalise (var i)     = i :: []
```

We also define a function `reify` to come back from a normal form to an expression.

```
reify : {n : Nat} -> NF n -> Expr n
reify []        = |∅|
reify (i :: nf) = var i |+| reify nf
```

We need decidable equality on normal forms, but since normal forms are justs lists of elements from a finite set we have already defined it.

```
nfDecEquiv : {n : Nat} -> DecidableEquivalence (NF n)
nfDecEquiv = listDecEquivalence finDecEquivalence
```

```
open module NfEq {n : Nat} =
    DecidableEquivalenceOps (nfDecEquiv {n})
  public
  using ()
  renaming ( Eq   to nfEquiv
           ; _==_ to _=NF=_
           )
```

We define equality on expressions to be equality on the corresponding normal forms.

```
exprDecEquiv : {n : Nat} -> DecidableEquivalence (Expr n)
exprDecEquiv = record
  { _==_ = \e1 e2 -> normalise e1 == normalise e2
  ; isEquiv = record
    { refl  = \e        -> refl  (normalise e)
    ; sym   = \e1 e2    -> sym   (normalise e1)(normalise e2)
    ; trans = \e1 e2 e3 ->
        trans (normalise e1)(normalise e2)(normalise e3)
    }
  ; decide = \e1 e2 -> decide (normalise e1)(normalise e2)
  }
  where
    open module NfEq = DecidableEquivalenceOps nfDecEquiv

open module EqExpr {n : Nat} =
    DecidableEquivalenceOps (exprDecEquiv {n})
  using    ()
  renaming ( _==_   to _=Expr=_
           ; decide to decideExprEq
           ; Eq     to exprEquiv
           )
```

This gives us a decidable equality on expressions and so we can decide whether or not an equation is provable simply by deciding the equality between the two sides. We create a datatype `IsProvable` recording provability. This is essentially the same type as returned by `decideExprEq` but with nicer names for the constructors.

```
data IsProvable {n : Nat} : Equation n -> Set where
  can-prove   : {e1 e2 : Expr n} ->
                e1 =Expr= e2     -> IsProvable (e1 ≐ e2)
  can't-prove : {e1 e2 : Expr n} ->
                ¬ (e1 =Expr= e2) -> IsProvable (e1 ≐ e2)
```

```
provable : {n : Nat}(thm : Equation n) -> IsProvable thm
provable (e₁ ≐ e₂) with decideExprEq e₁ e₂
provable (e₁ ≐ e₂) | inl p = can-prove p
provable (e₁ ≐ e₂) | inr p = can't-prove p
```

Note that we have not yet proved that our notion of provability is correct. That is the topic of the next module.

5.2.7 Semantics

Up until now we have not really done anything that could not be done in a simply typed language. We have defined a function to decide equality in a commutative monoid by flattening and sorting the expressions. What cannot be done in a simply typed setting is constructing the actual proof that the equation holds in any commutative monoid.

We define a module `Semantics` parameterised by an arbitrary commutative monoid.

```
open import Equivalence
open import Monoid

module Semantics {A : Set}{Eq : Equivalence A}
                 (M : CommutativeMonoid Eq)
  where

import Chain
open import Logic
open import Basics
open import Expr

private
  open module E = EquivalenceOps Eq
  open module M = CommutativeMonoidOps Eq M
  open module C = Chain _==_ refl trans
```

First, we have to define the semantics of an expression, i.e. how to translate it into an element of the monoid. To do this we need an environment containing values for the free variables of the expression.

```
Env : Nat -> Set
Env n = Vec A n
```

The semantic function replaces `|∅|` with ∅ and `|_+_|` with `_+_`. Variables are looked up in the environment (remember that `_!_` is the lookup function for vectors).

```
expr[_] : {n : Nat} -> Expr n -> Env n -> A
expr[ |∅|       ] ρ = ∅
expr[ e₁ |+| e₂ ] ρ = expr[ e₁ ] ρ + expr[ e₂ ] ρ
expr[ var i     ] ρ = ρ ! i
```

The semantics of a normal form is the semantics of the corresponding expression, and the semantics of an equation is the equivalence of the semantics of the expressions.

```
nf[_] : {n : Nat} -> NF n -> Env n -> A
nf[ xs ] ρ = expr[ reify xs ] ρ

eq[_] : {n : Nat} -> Equation n -> Env n -> Set
eq[ e₁ ≐ e₂ ] ρ = expr[ e₁ ] ρ == expr[ e₂ ] ρ
```

Now we can define what constitutes a proof of an equation. If the equation is provable a proof is a proof of the semantics of the equation for an arbitrary environment. If the equation is not provable no evidence is required and we simply demand an element of the singleton type `NoProof`. One could imagine providing a counter example in this case, but that would likely be more work than it is worth.

```
data NoProof : Set where
  no-proof : NoProof

Proof : {n : Nat} -> Equation n -> Set
Proof eq with provable eq
Proof (e₁ ≐ e₂) | can-prove p = (ρ : Env _) -> eq[ e₁ ≐ e₂ ] ρ
Proof (e₁ ≐ e₂) | can't-prove p = NoProof
```

In order to construct the proof of an equation we need to prove that our normalisation function is sound, i.e. that it preserves equality on the semantic side. First we prove that the merge function `_⊕_` is sound. The proof is straightforward and the equality reasoning parts are largely equations that could be proven using our algorithm.

```
⊕-sound : {n : Nat}(xs ys : NF n)(ρ : Env n) ->
          (nf[ xs ] ρ + nf[ ys ] ρ) == nf[ xs ⊕ ys ] ρ
⊕-sound []        ys        ρ = idL _
⊕-sound (x :: xs) []        ρ = idR _
```

```
⊕-sound (x :: xs) (y :: ys) ρ with x ⩽Fin⩽ y
⊕-sound (x :: xs) (y :: ys) ρ | true =
    chain> nf[ x :: xs ] ρ + nf[ y :: ys ] ρ
       === (ρ ! x + [xs]) + (ρ ! y + [ys])
           by refl _
       === ρ ! x + ([xs] + (ρ ! y + [ys]))
           by sym _ _ (assoc _ _ _)
       === ρ ! x + nf[ xs ⊕ (y :: ys) ] ρ
           by congL _ _ _ (⊕-sound xs (y :: ys) ρ)
       === nf[ x :: (xs ⊕ (y :: ys)) ] ρ
           by refl _
    qed
  where
    [xs] = nf[ xs ] ρ
    [ys] = nf[ ys ] ρ
⊕-sound (x :: xs) (y :: ys) ρ | false =
    chain> nf[ x :: xs ] ρ + nf[ y :: ys ] ρ
       === (ρ ! x + [xs]) + (ρ ! y + [ys])
           by refl _
       === (ρ ! y + [ys]) + (ρ ! x + [xs])
           by commute _ _
       === ρ ! y + ([ys] + (ρ ! x + [xs]))
           by sym _ _ (assoc _ _ _)
       === ρ ! y + ((ρ ! x + [xs]) + [ys])
           by congL _ _ _ (commute _ _)
       === ρ ! y + nf[ (x :: xs) ⊕ ys ] ρ
           by congL _ _ _ (⊕-sound (x :: xs) ys ρ)
       === nf[ y :: ((x :: xs) ⊕ ys) ] ρ
           by refl _
    qed
  where
    [xs] = nf[ xs ] ρ
    [ys] = nf[ ys ] ρ
```

It is worth pointing out that when we pattern match on `x ⩽Fin⩽ y` this expression is abstracted from the goal type, which makes the `if_then_else_` from `_⊕_` reduce.

Now proving that normalisation is sound is easy. In the variable case we add an extra $\emptyset$ so we have to use the axiom that $x + \emptyset = x$. The `|∅|` case is trivial and in the `_|+|_` case we use the fact that `_⊕_` is sound.

```
normalise-sound : {n : Nat}(e : Expr n)(ρ : Env n) ->
                  expr[ e ] ρ == nf[ normalise e ] ρ
normalise-sound (var i)     ρ = sym _ _ (idR _)
```

```
normalise-sound |∅|          ρ = refl _
normalise-sound (e₁ |+| e₂) ρ =
  chain> expr[ e₁ ] ρ + expr[ e₂ ] ρ
     === nf[ normalise e₁ ] ρ + nf[ normalise e₂ ] ρ
           by cong _ _ _ _ (normalise-sound e₁ ρ)
                           (normalise-sound e₂ ρ)
     === nf[ normalise e₁ ⊕ normalise e₂ ] ρ
           by ⊕-sound (normalise e₁) (normalise e₂) ρ
  qed
```

We also need a lemma stating that the equality on normal forms is sound.

```
nfEq-sound : {n : Nat}(xs ys : NF n)(ρ : Env n) ->
             xs =NF= ys -> nf[ xs ] ρ == nf[ ys ] ρ
nfEq-sound []       []          ρ eq = refl ∅
nfEq-sound []       (_ :: _)    ρ ()
nfEq-sound (_ :: _) []          ρ ()
nfEq-sound (x :: xs) (.x :: ys) ρ (ref , xsys) =
  congL _ _ _ (nfEq-sound xs ys ρ xsys)
```

Using our soundness lemmas we are now ready to define the function `prove` which takes an equation and computes a proof of it.

```
prove : {n : Nat}(eq : Equation n) -> Proof eq
prove eq with provable eq
prove (e₁ ≐ e₂) | can't-prove _ = no-proof
prove (e₁ ≐ e₂) | can-prove p   = \ ρ ->
  chain> expr[ e₁ ] ρ
     === nf[ n₁ ] ρ    by normalise-sound e₁ ρ
     === nf[ n₂ ] ρ    by nfEq-sound n₁ n₂ ρ p
     === expr[ e₂ ] ρ  by sym _ _ (normalise-sound e₂ ρ)
  qed
  where
    n₁ = normalise e₁
    n₂ = normalise e₂
```

Before giving any examples we define some functions to make our prover a little easier to use. The proof of a valid equation abstracts over an arbitrary environment, but a more natural result would be a curried version abstracting over the each element of the vector separately. We can define a `curry` function to translate into this form.

```
Curried : {A : Set}(n : Nat) -> (Vec A n -> Set) -> Set
Curried zero    P = P ε
Curried (suc n) P = (x : _) -> Curried n (\xs -> P (x ▶ xs))
```

```
curry : {A : Set}{n : Nat}{P : Vec A n -> Set} ->
          ((xs : Vec A n) -> P xs) -> Curried n P
curry {n = zero } f = f ε
curry {n = suc n} f = \x -> curry (\xs -> f (x ► xs))
```

For instance, given $P : \mathit{Vec}\ A\ 3 \to \mathit{Set}$ and $f : (\mathit{xs} : \mathit{Vec}\ A\ 3) \to P\ \mathit{xs}$ we have

$$\begin{aligned} \mathit{Curried}\ 3\ P &= (x\ y\ z : A) \to P\ (x \blacktriangleright y \blacktriangleright z \blacktriangleright \varepsilon) \\ \mathit{curry}\ f &= \lambda\, x\ y\ z \to f\ (x \blacktriangleright y \blacktriangleright z \blacktriangleright \varepsilon) \end{aligned}$$

Another thing which is tedious with the current presentation is to write down the equation to be proven. Since there is no way to reflect a goal type into an expression in our representation the equation has to be given explicitly. In order to save us the tedium of writing down the names of the free variables of an expression we can do a similar trick, only backwards. We define a type `_^_→_` of curried functions of the form $A \to \ldots \to A \to B$:

```
_^_→_ : Set -> Nat -> Set -> Set
A ^ zero  → B = B
A ^ suc n → B = A -> A ^ n → B
```

The `uncurry` function turns a curried function into an uncurried function.

```
uncurry : {A B : Set}{n : Nat} ->
          (A ^ n → B) -> (Vec A n -> B)
uncurry f ε        = f
uncurry f (x ► xs) = uncurry (f x) xs
```

Now we can define a function `equation` which given a function from n expressions to an equation over n variables applies the function to these variables. To get a vector of all free variables we simply tabulate the `var` function whose type is $\mathit{Fin}\ n \to \mathit{Expr}\ n$.

```
equation : (n : Nat) ->
           (Expr n ^ n → Equation n) -> Equation n
equation n eq = uncurry {n = n} eq (tabulate var)
```

Finally we are ready to put our prover to the test. As an example we prove part of the second case in the ⊕`-sound` proof. We use `curry` to get the result into the right form, and `equation` to make stating the equation easier.

```
test : (x xs y ys : A) ->
       ((x + xs) + (y + ys)) == (y + ((x + xs) + ys))
test = curry (prove eq)
  where
    eq = equation 4 \x xs y ys ->
          ((x |+| xs) |+| (y |+| ys)) ≐
          (y |+| ((x |+| xs) |+| ys))
```

It is still a bit inconvenient that the equation has to be stated twice, once in the monoid and once as an expression, but disregarding that it looks quite nice. A nice feature of this prover is that the proof does not have to be constructed. The only computation that needs to happen is deciding that the equation is provable, once that is done we know that `prove` gives us a valid proof, so it does not have to be built explicitly.

To avoid having to state the equation twice, we would need reflection, allowing us to inspect the structure of the goal type. This, however, is way beyond the scope of this thesis.

Chapter 6

First-order Logic

This chapter is based on the paper *Connecting a Logical Framework to a First-Order Logic Prover* [ACN05] written together with Andreas Abel and Thierry Coquand.

We present one way of combining a logical framework and first-order logic. The logical framework is used as an interface to a first-order theorem prover. The main purpose of the framework is to keep track of the structure of the proof and to deal with the high level steps, for instance, induction. The steps that involve purely propositional or simple first-order reasoning are left to a first-order resolution prover (the system Gandalf in our prototype). The correctness of this interaction is based on a general metatheoretic result. One feature is the simplicity of our translation between the logical framework and first-order logic. Implementation and case studies are described.

6.1 Introduction

We work towards human-readable and machine-verifiable *proof documents* for mathematics and computer science. As argued by de Bruijn [dB80], dependent type theory offers an ideal formal system for representing reasoning steps, such as introducing parameters or hypotheses, naming constants or lemmas, using a lemma or a hypothesis. Type theory provides explicit notations for these proof steps, with good logical properties. Using tools like Coq [BC04], Epigram [AMM05], or Agda [CC99] these steps can be performed interactively. But low level reasoning steps, such as simple propositional reasoning, or equality reasoning, substituting equals for equals, are tedious if performed in a purely interactive way. Furthermore, propositional provers, and even first-order logic (FOL) provers are now very efficient. It is thus natural to create interfaces between logical frameworks and automatic

propositional or first-order provers [BHdN02, ST95, MP04]. But, in order to arrive at proof documents which are still readable, only *trivial* proof steps should be handled by the automatic prover. Since different readers might have different notions of *trivial*, the automatic prover should not be a black box. With some effort by the human, the output of the prover should be understandable.

In this paper, we are exploring connections between a logical framework $\mathsf{MLF}_{\mathsf{Prop}}$ based on type theory and resolution-based theorem provers. One problem in such an interaction is that resolution proofs are hard to read and understand in general. Indeed, resolution proof systems work with formulæ in clause normal form, where clauses are (the universal closures of) disjunctions of literals, a literal being an atom or a negated atom. The system translates the negation of the statement to be proved to clause form, using skolemisation and disjunctive normal form. It then generates new clauses using resolution and paramodulation, trying to derive a contradiction. If successful, the system does pruning on the (typically high number of) generated clauses and outputs only the relevant ones.[1]

We lose the structure of the initial problem when doing skolemisation and clausification. Typically, a problem such as

$$\forall x.\exists y.\forall z.R(x,y) \Rightarrow R(x,z) \tag{1}$$

is negated and translated into the two contradictory unit clauses

$$\forall y.\ R(a,y), \qquad \forall y.\neg R(a,f(y)), \tag{2}$$

but the connection between the statement (1) and the refutation of (2) is not so intuitive.

We do not solve this problem here, but we point out that, if we restrict ourselves to implicitly universally quantified propositional formulæ, in the following called *open* formulæ, this problem does not arise. Furthermore, when we restrict to this fragment, we can use the idea of implicit typing [Bee07, WM89]. In this way, the translation from framework types to FOL formulæ is particularly simple. Technically, this is reflected by a general metatheorem which ensures that we can lift a first-order resolution proof to a framework derivation. If we restrict the class of formulæ further to so-called *geometrical* open formulæ [CLR01, BC03], then the translation to clausal form is transparent. Indeed, any resolution proof for this fragment is intuitionistically valid and can be interpreted as it is in type theory. This

[1] If the search is not successful, it is quite hard to get any relevant information from the clauses that are generated. We have not yet analyzed the problem of getting useful feedback in this case.

metatheorem is also the theoretical justification for our interface between $\mathsf{MLF}_{\mathsf{Prop}}$ and a resolution-based proof system.

We have implemented a prototype version of a type system in Haskell, with a connection to the resolution prover Gandalf [Tam97]. By restricting ourselves to open formulæ we sacrifice proof strength, but preliminary experiments show that the restriction is less severe than it may seem at first since the steps involving quantification are well handled at the framework level. Also, the proof traces produced by Gandalf are often readable (and surprisingly clever in some cases).

We think that we can represent Leslie Lamport proof style [Lam93] rather faithfully in this system. The high level steps such as introduction of hypotheses, case analysis, induction steps are handled at the framework level, and only the trivial steps are sent to the FOL automatic prover.

One can think of connecting the framework to other systems, e.g., rewriting systems and computer algebra systems. We have experimented with a connection to QuickCheck [CH00], that allows random testing of some propositions. In general, each connection extension of our logical framework should be justified in the same way as the one we present in this paper: we prove a conservativity result which ensures that the results from the external system can be, if desired, replaced by a direct proof in the framework. This way of combining various systems works in practice, as suggested by preliminary experiments, and it is theoretically well-founded.

This paper is organized as follows. We first describe the logical framework $\mathsf{MLF}_{\mathsf{Prop}}$. We then present the translation from some LF types to FOL formulæ. The main technical result is then a theorem that shows that any resolution and paramodulation step, with one restriction, can be lifted to the framework level. Finally, we present some examples and extensions, and a discussion of related work.

6.2 The Logical Framework $\mathsf{MLF}_{\mathsf{Prop}}$

This section presents an extension of Martin-Löf's logical framework [NPS00] by propositions and local definitions. This work was carried out in a different context than the work in previous chapters, and so uses a different logical framework. We believe, however, that the results carries over without great difficulty to UTT_{Σ} extended with propositions.

Expressions (terms and types). We assume countable sets of variables and constants. Furthermore, we have a finite number of built-in constants to construct the primitives of our type language. A priori, we do not distinguish

between terms and types. The syntactic entities of $\mathsf{MLF}_{\mathsf{Prop}}$ are given by the following grammar.

x, y, z			variables
c, f, p			constants
$\hat{c}$	$::=$	$\mathsf{Fun} \mid \mathsf{El} \mid \mathsf{Set} \mid () \mid \mathsf{Prf} \mid \mathsf{Prop}$	built-in constants
r, s, P, Q	$::=$	$\hat{c} \mid c \mid x \mid \lambda x.r \mid r\ s \mid \mathsf{let}\, x : T = r\, \mathsf{in}\, s$	expressions
T, U	$::=$	$\mathsf{Set} \mid \mathsf{El}\, s \mid \mathsf{Prop} \mid \mathsf{Prf}\, P \mid \mathsf{Fun}\, T\, (\lambda x.U)$	types
Γ	$::=$	$\diamond \mid \Gamma, x : T$	typing contexts
Σ	$::=$	$\diamond \mid \Sigma, c : T \mid \Sigma, c : T = r$	signatures

We identify terms and types up to α-conversion and adopt the convention that in contexts Γ, all variables must be distinct; hence, the context extension $\Gamma, x : T$ presupposes $x : U \notin \Gamma$ for any U. Similarly, a constant c may not be declared in a signature twice. We use the same syntactic conventions for UTT_Σ (see Section 1.3) and write $(x : T) \to U$ for $\mathsf{Fun}\, T\, (\lambda x.U)$.

The inhabitants of Set are type codes; El maps type codes to types. E. g., $(a : \mathsf{Set}) \to \mathsf{El}\, a \to \mathsf{El}\, a$ is the type of the polymorphic identity $\lambda a.\lambda x.x$. Similarly Prop contains formal propositions P and $\mathsf{Prf}\, P$ proofs of P.

Types of the shape $\Gamma \to \mathsf{Prf}\, P$ are called *proof types*. A context $\Gamma = (x_1 : T_1) \ldots (x_n : T_n)$ is a *set context* if and only if all T_i are of the form $\Delta \to \mathsf{El}\, S$. In particular, if $P : \mathsf{Prop}$, then the proof type $\Gamma \to \mathsf{Prf}\, P$ corresponds to a universal first-order formula $\forall x_1 \ldots \forall x_n P$ with quantifier-free kernel P.

Judgements. The type theory $\mathsf{MLF}_{\mathsf{Prop}}$ is presented via five judgements, which are all relative to a (user-defined) signature Σ.

$\Gamma \vdash_\Sigma$	Γ is a well-formed context
$\Gamma \vdash_\Sigma T$	T is a well-formed type
$\Gamma \vdash_\Sigma r : T$	r has type T
$\Gamma \vdash_\Sigma T = T'$	T and T' are equal types
$\Gamma \vdash_\Sigma r = r' : T$	r and r' are equal terms of type T

All five judgements are defined simultaneously. Since the signature remains fixed in all judgements we will omit it.

Judgmental type and term equality are generated from expansion of signature definitions as well as from β-, η-, and let-equality, the latter of which is given by $(\mathsf{let}\, x : T = r\, \mathsf{in}\, s) = s[x := r]$. The rules for equality are similar to the ones of MLF_Σ [AC05], and type-checking of normal terms with local definitions is decidable.

Figure 6.1 shows the typing rules. The rules FUN-F and FUN-I carry a side condition $(*)$ that ensures that no type can depend on a proof, which is needed for the conservativity theorem.

Wellformed contexts $\Gamma \vdash$.

$$\textsc{cxt-empty}\ \frac{}{\diamond \vdash} \qquad \textsc{cxt-ext}\ \frac{\Gamma \vdash T}{\Gamma, x : T \vdash}$$

Wellformed types $\Gamma \vdash T$.

$$\textsc{set-f}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Set}} \qquad \textsc{prop-f}\ \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Prop}} \qquad \textsc{fun-f}\ \frac{\Gamma \vdash T \qquad \Gamma, x : T \vdash U}{\Gamma \vdash (x : T) \to U}\ (*)$$

$$\textsc{set-e}\ \frac{\Gamma \vdash r : \mathsf{Set}}{\Gamma \vdash \mathsf{El}\ r} \qquad \textsc{prop-e}\ \frac{\Gamma \vdash P : \mathsf{Prop}}{\Gamma \vdash \mathsf{Prf}\ P}$$

Typing $\Gamma \vdash r : T$.

$$\textsc{cst}\ \frac{\Gamma \vdash \quad (c : T) \in \Sigma}{\Gamma \vdash c : T} \qquad \textsc{hyp}\ \frac{\Gamma \vdash \quad (x : T) \in \Gamma}{\Gamma \vdash x : T}$$

$$\textsc{conv}\ \frac{\Gamma \vdash r : T \quad \Gamma \vdash T = U}{\Gamma \vdash r : U} \qquad \textsc{fun-i}\ \frac{\Gamma, x : T \vdash r : U}{\Gamma \vdash \lambda x.r : (x : T) \to U}\ (*)$$

$$\textsc{fun-e}\ \frac{\Gamma \vdash r : (x : T) \to U \qquad \Gamma \vdash s : T}{\Gamma \vdash r\ s : U[x := s]}$$

$$\textsc{let}\ \frac{\Gamma \vdash r : T \qquad \Gamma \vdash s[x := r] : U}{\Gamma \vdash \mathsf{let}\ x : T = r\ \mathsf{in}\ s : U}$$

Side condition $(*)$: If T is a proof type, then also U.

Figure 6.1: $\mathsf{MLF}_{\mathsf{Prop}}$ rules for contexts and typing.

Natural deduction. We assume a signature Σ_{nd} given in Figure 6.2, which assumes the infix logical connectives $op ::= \wedge, \vee, \Rightarrow$, plus the defined ones, $\neg$ and $\Leftrightarrow$. Furthermore, it contains a set $\mathsf{PredSym}$ of basic predicate symbols p of type $\Gamma \to \mathsf{Prop}$ where Γ is a (possibly empty) set context. Currently we only assume truth $\top$, absurdity $\bot$, and typed equality Id, but user defined signatures can extend $\mathsf{PredSym}$ by their own symbols. For each logical constructs, there are appropriate proof rules, e. g., a constant $\mathsf{impI} : (P, Q : \mathsf{Prop}) \to (\mathsf{Prf}\ P \to \mathsf{Prf}\ Q) \to \mathsf{Prf}\ (P \Rightarrow Q)$.

First-order logic assumes that every set is non-empty, and our use of a first-order prover is only sound under this assumption. Hence, we add a special constant $\epsilon : (D : \mathsf{Set}) \to \mathsf{El}\ D$ to Σ_{nd} which enforces this fact. Notice that this implies that all set contexts are inhabited[2].

Classical reasoning can be performed in the signature Σ_{class}, which we define as the extension of Σ_{nd} by $\mathsf{EM} : (P : \mathsf{Prop}) \to \mathsf{Prf}\ (P \vee \neg P)$, the law of the excluded middle.

The FOL rule. This article investigates conditions under which the addition of the following rule is conservative over $\mathsf{MLF}_{\mathsf{Prop}} + \Sigma_{\mathsf{nd}}$ and $\mathsf{MLF}_{\mathsf{Prop}} + \Sigma_{\mathsf{class}}$, respectively.

$$\textsc{fol}\ \frac{\Gamma \vdash T}{\Gamma \vdash () : T}\ \Gamma \vdash_{\mathsf{FOL}} T$$

The side condition $\Gamma \vdash_{\mathsf{FOL}} T$ expresses that T is a proof type and that the first-order prover can deduce the truth of the corresponding first-order formula from the assumptions in Γ. It ensures that only tautologies have proofs in $\mathsf{MLF}_{\mathsf{Prop}}$, but it is not considered part of the type checking. Metatheoretical properties of $\mathsf{MLF}_{\mathsf{Prop}}$ like decidability of equality and type-checking hold independently of this side condition.

Conservativity fails if we have to compare proof objects during type-checking. This is because the rule FOL produces a single proof object for all (true) propositions, whereas upon removal of FOL the hole has to be filled with specific proof object. Hence two equal objects which each depend on a proof generated by FOL could become unequal after replacing FOL. To avoid this, it is sufficient to restrict function spaces $(x : T) \to U$: if T is a proof type, then also U. While this restriction is clearly sufficient, it is rather sever. For instance, it is not possible to define a function computing an element of a set under some propositional preconditions. What we really need here is proof irrelevant propositions.

In the remainder of the paper, we use LF as a synonym for $\mathsf{MLF}_{\mathsf{Prop}}$.

[2]Semantically, it may be fruitful to think of terms of type Set as inhabited Partial Equivalence Relations, while terms of type Prop are PERs with at most one inhabitant.

Predicate symbols and logical connectives.

$$\begin{array}{llll} \mathsf{PredSym} & \ni p & ::= & \top, \bot, Id \quad \text{predicate symbols} \\ \mathsf{LogOp} & \ni op & ::= & \wedge, \vee, \Rightarrow \quad \text{binary logical connectives} \end{array}$$

Formation rules for propositional logic.

$$\begin{array}{lll} \top, \bot & : \mathsf{Prop} & \text{truth, absurdity} \\ \wedge, \vee, \Rightarrow & : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} & \text{conj., disj., impl.} \\ \neg & : \mathsf{Prop} \to \mathsf{Prop} \qquad = \lambda P.\, P \Rightarrow \bot & \text{negation} \\ \Leftrightarrow & : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop} & \\ & = \lambda P \lambda Q.\, (P \Rightarrow Q) \wedge (Q \Rightarrow P) & \text{equivalence} \end{array}$$

Proof rules for propositional logic.

$$\begin{array}{lll} \mathsf{truel} & : \mathsf{Prf}\, \top & \\ \mathsf{falseE} & : (P : \mathsf{Prop}) \to \mathsf{Prf}\, \bot \to \mathsf{Prf}\, P & \\ \\ \mathsf{andI} & : (P_1, P_2 : \mathsf{Prop}) \to \mathsf{Prf}\, P_1 \to \mathsf{Prf}\, P_2 \to \mathsf{Prf}\, (P_1 \wedge P_2) & \\ \mathsf{andE}_i & : (P_1, P_2 : \mathsf{Prop}) \to \mathsf{Prf}\, (P_1 \wedge P_2) \to \mathsf{Prf}\, P_i & \text{for } i \in \{1, 2\} \\ \\ \mathsf{orI}_i & : (P_1, P_2 : \mathsf{Prop}) \to \mathsf{Prf}\, P_i \to \mathsf{Prf}\, (P_1 \vee P_2) & \text{for } i \in \{1, 2\} \\ \mathsf{orE} & : (P_1, P_2, Q : \mathsf{Prop}) \to \mathsf{Prf}\, (P_1 \vee P_2) \to \\ & \quad (\mathsf{Prf}\, P_1 \to \mathsf{Prf}\, Q) \to (\mathsf{Prf}\, P_2 \to \mathsf{Prf}\, Q) \to \mathsf{Prf}\, Q & \\ \\ \mathsf{impl} & : (P, Q : \mathsf{Prop}) \to (\mathsf{Prf}\, P \to \mathsf{Prf}\, Q) \to \mathsf{Prf}\, (P \Rightarrow Q) & \\ \mathsf{impE} & : (P, Q : \mathsf{Prop}) \to \mathsf{Prf}\, (P \Rightarrow Q) \to \mathsf{Prf}\, P \to \mathsf{Prf}\, Q & \end{array}$$

Equality.

$$\begin{array}{lll} Id & : (D : \mathsf{Set}) \to \mathsf{El}\, D \to \mathsf{El}\, D \to \mathsf{Prop} & \text{typed equality} \\ \mathsf{refl} & : (D : \mathsf{Set},\ x : \mathsf{El}\, D) \to \mathsf{Prf}\, (Id\, D\, x\, x) & \text{reflexivity} \\ \mathsf{subst} & : (D : \mathsf{Set},\ P : \mathsf{El}\ D \to \mathsf{Prop},\ x, y : \mathsf{El}\ D) \to & \\ & \quad \mathsf{Prf}\, (Id\, D\, x\, y) \to \mathsf{Prf}\, (P\, x) \to \mathsf{Prf}\, (P\, y) & \text{substitutivity} \end{array}$$

Figure 6.2: The signature Σ_{nd} for natural deduction.

6.3 Translation from $\mathsf{MLF}_{\mathsf{Prop}}$ to FOL

We shall define a *partial* translation from some LF types to FOL propositions. We translate only types of the form

$$(x_1 : T_1) \ldots (x_k : T_k) \to \mathsf{Prf}\ (P(x_1, \ldots, x_k)),$$

and these are translated to *open* formulæ $[P(x_1, \ldots, x_k)]$ of first-order logic. All the variables $x_1, \ldots, x_k$ are considered universally quantified. For instance,

$$(x : \mathsf{El}\ \mathit{Nat}) \to \mathsf{Prf}\ (\mathit{Id}\ \mathit{Nat}\ x\ x \wedge \mathit{Id}\ \mathit{Nat}\ x\ (\mathit{add}\ \mathsf{zero}\ x))$$

will be translated to $x = x\ \wedge\ x = \mathit{add}\ \mathsf{zero}\ x$. If we have a theory of lattices, that is, we have added

$$\begin{array}{lcl} D & : & \mathsf{Set} \\ \mathit{sup} & : & \mathsf{El}\ D \to \mathsf{El}\ D \to \mathsf{El}\ D \\ \leqslant & : & \mathsf{El}\ D \to \mathsf{El}\ D \to \mathsf{Prop} \end{array}$$

to the current signature, then $(x, y : \mathsf{El}\ D) \to \mathsf{Prf}\ (\mathit{sup}\ x\ y \leqslant x \Leftrightarrow y \leqslant x)$ would be translated to $\mathit{sup}\ x\ y \leqslant y\ \Leftrightarrow\ y \leqslant x$.

The translation is done at a syntactical level, without using types. We demonstrate that we can lift a resolution proof of a translated formula to an LF derivation in the signature Σ_{class} (or in Σ_{nd}, in some cases).

6.3.1 Formal Description of the Translation

We translate *normal* expressions, which means that all definitions have been unfolded and all redexes reduced. Three classes of normal $\mathsf{MLF}_{\mathsf{Prop}}$-expressions are introduced: (formal) *first-order terms* and (formal) *first-order formulæ*, which are quantifier free formulæ over atoms possibly containing free term variables, and *translatable formulæ*, which are first-order formulæ prefixed by quantification over set elements.

$$\begin{array}{llll} t, u & ::= & x \mid f\,\vec{t} & \text{first-order terms} \\ A, B & ::= & p\,\vec{t} \mid \mathit{Id}\,S\,t_1\,t_2 & \text{atoms} \\ W & ::= & A \mid W\ \mathit{op}\ W' & \text{first-order formulæ} \\ \phi & ::= & \Delta \to \mathsf{Prf}\,W & \text{translatable formulæ } (\Delta \text{ set context}) \end{array}$$

Proper terms are those which are not just variables. For the conservativity result the following fact about proper terms will be important: In a well-typed proper term, the types of its variables are uniquely determined. For

this reason, a formal first-order term t may neither contain a binder (λ or let) nor a variable which is applied to something, for instance, $x\,u$.

An example of a first-order formula is

$$W_{ex} := \mathit{Id}\ D\ x\ (f\ y) \Rightarrow (\mathit{Less}\ x\ (f\ y) \Rightarrow \bot)$$

which is well-typed in the extension

$$\begin{array}{lll} D & : & \mathsf{Set} \\ f & : & \mathsf{El}\ D \ \to\ \mathsf{El}\ D \\ \mathit{Less} & : & \mathsf{El}\ D \ \to \mathsf{El}\ D \ \to\ \mathsf{Prop} \end{array}$$

of the signature Σ_{nd}.

On the FOL side, we consider a language with equality ($=$), one binary function symbol app and one constant for each constant introduced in the logical framework. Having an explicit "app" allows partial application of function symbols.

Let $\Delta = (x_1 : T_1) \ldots (x_n : T_n)$ be a set context. A type of the form

$$\phi := \Delta \to \mathsf{Prf}\ W$$

is translated into a universal formula $[\phi] = \forall x_1 \ldots \forall x_n [W]$. The translation $[W]$ of first-order formulæ and the translation $\langle t \rangle$ of first-order terms depends on Δ and is defined recursively as follows:

$$\begin{array}{lcll} [W_1\ \mathit{op}\ W_2] & := & [W_1]\ \mathit{op}\ [W_2] & \text{logical connectives} \\ [\mathit{Id}\ S\ t_1\ t_2] & := & \langle t_1 \rangle = \langle t_2 \rangle & \text{equality} \\ [p\ t_1 \ldots t_n] & := & p(\langle t_1 \rangle, \ldots, \langle t_n \rangle) & \text{predicates, including } \top, \bot \\ \langle x_i \rangle & := & x_i & x_i \in \Delta \\ \langle x \rangle & := & c_x & x \notin \Delta \\ \langle c \rangle & := & c & \text{constants} \\ \langle f\ t_1 \ldots t_n \rangle & := & f(\langle t_1 \rangle, \ldots, \langle t_n \rangle) & \text{n-ary functions} \end{array}$$

where we write $f(t_1, \ldots, t_n)$ for $\mathsf{app}(\ldots \mathsf{app}(\mathsf{app}(f, t_1), t_2), \ldots, t_n)$. Note that the translation is purely syntactical, and does not use type information. It is even homomorphic with two exceptions: (a) the typed equality of $\mathsf{MLF}_{\mathsf{Prop}}$ is translated into the untyped equality of FOL, and (b) variables bound outside ϕ have to be translated as constants.

For instance, the formula

$$(y : \mathsf{El}\ D) \to \mathit{Id}\ D\ x\ (f\ y) \Rightarrow (\mathit{Less}\ x\ (f\ y) \Rightarrow \bot)$$

is translated to

$$\forall y.\ c_x = f(y) \Rightarrow (\mathit{Less}(c_x, f(y)) \Rightarrow \bot)$$

Examples of types that cannot be translated are

$(x : \mathsf{Prop}) \to \mathsf{Prf}\ x$	$(x : \mathsf{Prop})$ is not a set context
$\mathsf{Prf}\ (F\ (\lambda x.x))$	$\lambda x.x$ is not a first-order term
$(y : \mathsf{El}\ D \to \mathsf{El}\ D) \to \mathsf{Prf}\ (P\ (y\ x))$	$y\ x$ is not a first-order term

We shall also use the class of *geometrical formulæ*, given by the following grammar:

$$\begin{array}{lcll} G & ::= & H \mid H \to G \mid G \wedge G & \text{geometrical formula} \\ H & ::= & A \mid H \wedge H \mid H \vee H & \text{positive formula} \end{array}$$

The above example W_{ex} is geometrical. As we will show, (classical) first-order proofs of geometrical formulæ can be mapped to intuitionistic proofs in the logical framework with Σ_{nd}.

6.3.2 Resolution Calculus

It will be convenient to use the following non-standard presentation of the resolution calculus [Rob65]. A *clause* C is an open first-order formula of the form

$$A_1 \wedge \cdots \wedge A_n \Rightarrow B_1 \vee \cdots \vee B_m$$

where we can have $n = 0$ or $m = 0$ and A_i and B_j are atomic formulæ. Following Gentzen [Gen35], we write such a clause on the form

$$A_1, \ldots, A_n \Rightarrow B_1, \ldots, B_m,$$

that is, $X \Rightarrow Y$, where X and Y are finite sets of atomic formulæ. An empty X is interpreted as truth, an empty Y as absurdity.

Resolution is forward reasoning. Figure 6.3 lists the rules for extending the current set of derived clauses: if all clauses mentioned in the premise of a rule are present, this rule can fire and the clause of the conclusion is added to the clause set.

In our formulation, all rules are intuitionistically valid[3], and can be justified in $\mathsf{MLF}_{\mathsf{Prop}} + \Sigma_{\mathsf{nd}}$. It can be shown, classically, that these rules are *complete* in the following sense: if a clause is a semantical consequence of other clauses then it is possible to derive it using the resolution calculus. Hence, any proof in FOL can be performed with resolution[4].

It can be pointed out that the SUB rule is only necessary at the very end—any resolution proof can be normalized to a proof that only uses SUB in the final step.

[3] In the standard formulation, the AX rule would read $\neg A \vee A$—the excluded middle.

[4] To deal with existential quantification we also need skolemisation.

$$\textsc{ax}\ \frac{}{A \Rightarrow A} \qquad \textsc{sub}\ \frac{X' \supseteq X \qquad X \Rightarrow Y \qquad Y \subseteq Y'}{X' \Rightarrow Y'}$$

$$\textsc{res}\ \frac{X_1 \Rightarrow Z_1, Y_1 \qquad X_2, Z_2 \Rightarrow Y_2}{(X_1, X_2 \Rightarrow Y_1, Y_2)\sigma}\ \sigma = \mathsf{mgu}(Z_1, Z_2)$$

$$\textsc{refl}\ \frac{}{\cdot \Rightarrow x = x}$$

$$\textsc{para}\ \frac{X_1 \Rightarrow t = u, Y_1 \qquad X_2[t'] \Rightarrow Y_2[t']}{(X_1, X_2[u] \Rightarrow Y_1, Y_2[u])\sigma}\ \sigma = \mathsf{mgu}(t, t')$$

Figure 6.3: Resolution calculus.

Let the *restricted* paramodulation rule denote the version of PARA where both t and t' are proper terms (not variables). The restricted rule is needed to preserve well-typedness.

6.3.3 Proof of Correctness

In this section, we show that every FOL proof of a translated formula $[\phi]$ can be lifted to a proof in $\mathsf{MLF}_{\mathsf{Prop}} + \Sigma_{\mathsf{class}}$, provided the resolution proof confines to restricted paramodulation. This is not trivial because FOL is untyped and $\mathsf{MLF}_{\mathsf{Prop}}$ is typed, and our translation forgets the types. The crucial insight is that every resolution step preserves well-typedness.

Fix a signature Σ. A first-order term t is *well-typed* if and only if there exists a context Δ, giving types to the variables $x_1, \ldots, x_n$ of t, such that in the given signature, $\Delta \vdash t : T$ for some type T. For example, in the signature

$$\begin{array}{lcllcl} D & : & \mathsf{Set} & f & : & \mathsf{El}\ D \to \mathsf{El}\ D \\ F & : & \mathsf{El}\ D \to \mathsf{Prop} & g & : & (x : \mathsf{El}\ D) \to \mathsf{Prf}\ (F\ x) \end{array}$$

the proper first-order terms $f\ x$, $F\ y$, and $g\ z$ are well-typed, but $F\ x\ y$ is not. Notice that if a *proper* FOL term is well-typed, then there is only one way to assign types to its variables.

We say that the terms $t_1, \ldots, t_n$ *fit* a context $\Delta = (x_1 : T_1) \ldots (x_n : T_n)$ in Γ if and only if $\Gamma \vdash t_i : T_i[t_1, \ldots, t_{i-1}]$ for all $1 \leqslant i \leqslant n$.

Lemma 6.3.1. *If two proper first-order terms t_1, t_2 over disjoint variables are well-typed and unifiable, then the most general unifier $\mathsf{mgu}(t_1, t_2)$ is well-typed.*

Proof. The lemma is a consequence of the following stronger proposition: *If* $t_1, \ldots, t_n$ *and* $u_1, \ldots, u_n$ *are lists of terms that fit the same context* Δ *in* Γ *and* σ *is the most general substitution such that* $t_i\sigma = u_i\sigma$ *for* $1 \leqslant i \leqslant n$*, then* $\Gamma \vdash \sigma(x) : A$ *for all* $(x : A) \in \Gamma$.

Let $\Gamma \vdash t : A$ and $\Gamma' \vdash u : B$. Since t and u are proper terms and unifiable, $t = f(\vec{t})$ and $u = f(\vec{u})$ for some constant $f : \Delta \to C$. Hence, $\vec{t}$ and $\vec{u}$ fit Δ in $\Gamma\Gamma'$, which is a valid context since Γ and Γ' are disjoint. Now the proposition implies that $\mathsf{mgu}(t, u)$ is well-typed.

To prove the stronger proposition, we follow the steps of a simple unification algorithm and consider the unification problem

$$t_1 = u_1, \ \ldots, \ t_n = u_n$$

If both t_1 and u_1 are proper terms, they are of the form $f(a_1, \ldots, a_k)$ and $f(b_1, \ldots, b_k)$ and we get a simpler unification problem

$$a_1 = b_1, \ \ldots, \ a_k = b_k, \ t_2 = u_2, \ \ldots, \ t_n = u_n$$

If, for instance, t_1 is a variable x, and x does not appear in u_1, we claim that all variables in u_1 have a type which is independent of x. This holds if u_1 is a variable, since the type of u_1 is the same as the one of x, but it also holds if u_1 is a proper term, since the type of the variables in u_1 are then determined by u_1 alone, and x does not appear in u_1. We can hence assume that all these variables appear before x in $\Gamma = \Gamma_1, x : T, \Gamma_2$. We then get the simpler unification problem in $\Gamma_1, \Gamma_2[x := u_1]$

$$t_2[x := u_1] = u_2[x := u_1], \ \ldots, \ t_n[x := u_1] = u_n[x := u_1]$$

We proceed in this way until we get an empty list in the context in which the most general unifier of the two terms is well-typed. □

For instance, *add* x zero and *add* (suc y) z are unifiable and well-typed and the most general unifier $\{x \mapsto \mathsf{suc}\, y, z \mapsto \mathsf{zero}\}$ is well-typed.

Using this lemma, we can lift any FOL resolution step to an LF resolution step. The same holds for any *restricted* paramodulation step, which justifies the translation of *Id* S t u as $\langle t \rangle = \langle u \rangle$ in FOL, Indeed, in the paramodulation step between $X_1 \Rightarrow t = u, Y_1$ and $X_2[t'] \Rightarrow Y_2[t']$ we unify t and t' and for Lemma 6.3.1 to be applicable both t and t' have to be proper terms. Similar arguments have been put forth by Beeson [Bee07] and Wick and McCune [WM89].

A clausal type is a formula which translates to a clause.

Lemma 6.3.2. *If two FOL clausal types* $\Gamma_1 \to \mathsf{Prf}\ (W_1)$ *and* $\Gamma_2 \to \mathsf{Prf}\ (W_2)$ *are derivable, and* C *is a resolution of* $[W_1]$ *and* $[W_2]$ *then there exists a context* Γ *and a derivable* $(\Gamma) \to \mathsf{Prf}\ W$ *such that* $C = [W]$*. The same holds if* C *is derived from* $[W_1]$ *and* $[W_2]$ *by restricted paramodulation. Furthermore in both cases,* Γ *is a set context if both* Γ_1 *and* Γ_2 *are set contexts.*

Proof. Using Lemma 6.3.1 in the cases where unification is performed. □

In the next theorems, $\phi, \phi_1, \ldots, \phi_k$ are translatable formulæ of the form $\Gamma \to \mathsf{Prf}\ W$ where Γ is a set context.

The following theorem is a consequence of Lemma 6.3.2, since an open formula is (classically) equivalent to a conjunction of clauses.

Theorem 6.3.3. *If we can derive* $[\phi]$ *from* $[\phi_1], \ldots, [\phi_k]$ *by resolution and restricted paramodulation then* ϕ *is derivable from* $\phi_1, \ldots, \phi_k$ *in any extension of the signature* Σ_{class}*.*

Proof. By induction on the derivation, using Lemma 6.3.2 in each step. □

A resolution proof, as we have presented it, is intuitionistically valid. The only step which may not be intuitionistically valid is when we express the equivalence between an open formula and a conjunction of clauses. For instance the open formula $\neg P \vee Q$ is not intuitionistically equivalent to the clause $P \Rightarrow Q$ in general. This problem does not occur if we start with geometrical formulæ [BC03].

Theorem 6.3.4. *If we can derive* $[\phi]$ *from* $[\phi_1], \ldots, [\phi_k]$ *by resolution and restricted paramodulation and* $\phi, \phi_1, \ldots, \phi_k$ *are geometric formulæ, then* ϕ *is derivable from* $\phi_1, \ldots, \phi_k$ *in any extension of the signature* Σ_{nd}*.*

Proof. Follows from the fact that clausification is intuitionistically valid for geometric formulæ. □

It is important for the theorem that all set contexts are inhabited: if $D : \mathsf{Set}$ and $P : \mathsf{Prop}$ (with x not free in P), then both

$$\phi_1 = (x : \mathsf{El}\ D) \to \mathsf{Prf}\ P \quad \text{and} \quad \phi_2 = \mathsf{Prf}\ P$$

are translated to the same FOL proposition $[\phi_1] = [\phi_2] = P$ but we can derive ϕ_2 from ϕ_1 in $\Sigma_{\mathsf{nd}}, D : \mathsf{Set}, P : \mathsf{Prop}$ only because $\mathsf{El}\ D$ is inhabited.

As noticed above, if we allow paramodulation from a variable, we could derive clauses that are not well-typed. For instance, in the signature

$$\begin{array}{lcl} Nat & : & \mathsf{Set} \\ \mathsf{zero} & : & \mathsf{El}\ Nat \\ h & : & (x : \mathsf{El}\ Nat) \to \mathsf{Prf}\ (Id\ Nat\ x\ \mathsf{zero}) \\ A & : & \mathsf{Set} \\ a & : & \mathsf{El}\ A \end{array}$$

the type of h becomes $x = \mathsf{zero}$ in FOL and from this we could derive, by paramodulation from the variable x, $a = \mathsf{zero}$ which is not well-typed. This problem is also discussed in [Bee07, WM89] and the solution is simply to forbid the FOL prover to use paramodulation from a variable[5].

We can now state the conservativity theorem.

Theorem 6.3.5. *If a type is inhabited in the system* $\mathsf{MLF}_{\mathsf{Prop}} + \textsc{fol} + \Sigma_{\mathsf{class}}$ *then it is inhabited in* $\mathsf{MLF}_{\mathsf{Prop}} + \Sigma_{\mathsf{class}}$.

Proof. By induction on the typing derivation, using Theorem 6.3.3 for FOL derivations. □

6.3.4 Simple Examples

Figure 6.4 shows an extension of Σ_{nd} by natural numbers, induction and an addition function defined by recursion on the second argument. Now

Nat	:	Set	natural numbers
zero	:	$\mathsf{El}\, Nat$	zero
suc	:	$\mathsf{El}\, Nat \to \mathsf{El}\, Nat$	successor
$indNat$	:	$(P : \mathsf{El}\, Nat \to \mathsf{Prop}) \to P\ \mathsf{zero} \to$ $((x : \mathsf{El}\, Nat) \to P\ x \Rightarrow P\ (\mathsf{suc}\ x)) \to$ $(n : \mathsf{El}\, Nat) \to P\ n$	induction
add	:	$\mathsf{El}\, Nat \to \mathsf{El}\, Nat \to \mathsf{El}\, Nat$	addition
$addZero$	:	$(x : \mathsf{El}\, Nat) \to Id\ Nat\ (add\ x\ \mathsf{zero})\ x$	axiom 1 of add
$addSuc$	:	$(x, y : \mathsf{El}\, Nat) \to$ $Id\ Nat\ (add\ x\ (\mathsf{suc}\ y))\ (\mathsf{suc}\ (add\ x\ y))$	axiom 2 of add

Figure 6.4: A signature of natural numbers and addition.

consider the goal $(x : \mathsf{El}\, Nat) \to Id\ Nat\ (add\ \mathsf{zero}\ x)\ x$. Using the induction

[5]This is possible in Otter. In Gandalf, this could be checked from the trace. Paramodulation from a variable is highly non-deterministic. For efficiency reasons, it was not present in some version of Gandalf, but it was added later for completeness. In the examples we have tried, this restriction is not a problem.

schema and the propositional proof rules, we can give the proof term

$$indNat\ (\lambda x.\ Id\ Nat\ (add\ \mathsf{zero}\ x)\ x)\ ()\ (\lambda a.\ \mathsf{impl}\ (\lambda ih\ ()))$$

in the logical framework, which contains these two FOL goals:

$$\begin{array}{c} \vdash_{\mathsf{FOL}} Id\ Nat\ (add\ \mathsf{zero}\ \mathsf{zero})\ \mathsf{zero} \\ (a : \mathsf{El}\ Nat)(ih : Id\ Nat\ (add\ \mathsf{zero}\ a)\ a) \vdash_{\mathsf{FOL}} \\ Id\ Nat\ (add\ \mathsf{zero}\ (\mathsf{suc}\ a))\ (\mathsf{suc}\ a) \end{array}$$

Both goals can be handled by the FOL prover. The first goal becomes $add\ \mathsf{zero}\ \mathsf{zero} = \mathsf{zero}$ and is proved from $add\ x\ \mathsf{zero} = x$, the translation of axiom *addZero*. The second goal becomes $add\ \mathsf{zero}\ (\mathsf{suc}\ a) = \mathsf{suc}\ a$. This is a first-order consequence of the translated induction hypothesis $add\ \mathsf{zero}\ a = a$ and $add\ x\ (\mathsf{suc}\ y) = \mathsf{suc}\ (add\ x\ y)$, the translation of axiom *addSuc*.

This example, though very simple, is a good illustration of the interaction between LF and FOL: the framework is used to handle the induction step and in the second goal, the introduction of the parameter a and the induction hypothesis.

Here is another simple example which illustrates that we can call the FOL prover even in a context involving non first-order operations. This example comes from a correctness proof of Warshall's algorithm. Let $D : \mathsf{Set}$.

$$\begin{array}{l} F : \mathsf{El}\ D \to (\mathsf{El}\ D \to \mathsf{El}\ D \to \mathsf{Prop}) \to \mathsf{El}\ D \to \mathsf{El}\ D \to \mathsf{Prop} \\ F\ a\ R\ x\ y = R\ x\ y \vee (R\ x\ a \wedge R\ a\ y) \\ \\ swap : (abxy : \mathsf{El}\ D) \to \mathsf{Prf}\ (F\ a\ (F\ b\ R)\ x\ y \Leftrightarrow F\ b\ (F\ a\ R)\ x\ y) \end{array}$$

The operation F is a higher-order operation. However, in the context $R : \mathsf{El}\ D \to \mathsf{El}\ D \to \mathsf{Prop}$, the goal *swap* can be handled by the FOL prover. The normal form of $F\ a\ (F\ b\ R)\ x\ y \Leftrightarrow F\ b\ (F\ a\ R)\ x\ y$, where all defined constants (here only F) have been unfolded, is a translatable formula.

6.4 Implementation

To try out the ideas described in this paper we have implemented a prototype type checker [Nor06] in Haskell. In addition to the logical framework, the type checker supports implicit arguments and the extensions described in Section 6.7: sigma types, datatypes and definitions by pattern matching. Note that this implementation is not the same as the Agda language from Chapter 5.

6.4.1 Implicit Arguments

A problem with LF as presented here is its rather heavy notation. For instance, to state that function composition is associative one would give the signature in Figure 6.5. This is very close to being completely illegible

---

$$
\begin{array}{l}
comp : (A\ B\ C : \mathsf{Set}) \to (\mathsf{El}\ B \to \mathsf{El}\ C) \to (\mathsf{El}\ A \to \mathsf{El}\ B) \to (\mathsf{El}\ A \to \mathsf{El}\ C) \\
comp\ A\ B\ C\ f\ g = \lambda x.\ f\ (g\ x)
\end{array}
$$

$$
\begin{array}{ll}
assoc : & (A\ B\ C\ D : \mathsf{Set}) \to \\
& (f : \mathsf{El}\ C \to \mathsf{El}\ D,\ g : \mathsf{El}\ B \to \mathsf{El}\ C,\ h : \mathsf{El}\ A \to \mathsf{El}\ B) \to \\
& \mathsf{Prf}\ (Id\ (\mathsf{El}\ A \to \mathsf{El}\ D)\ \ (comp\ A\ C\ D\ f\ (comp\ A\ B\ C\ g\ h)) \\
& \qquad\qquad\qquad\qquad\ \ (comp\ A\ B\ D\ (comp\ B\ C\ D\ f\ g)\ h))
\end{array}
$$

Figure 6.5: Associativity without Implicit Arguments.

due to the fact that we have to be explicit about the type arguments to the composition function. To solve the problem, we have implemented a mechanism for implicit arguments which allows the omission of arguments that can be inferred automatically (see Chapter 3). Using this mechanism the associativity example can be written as follows:

$$
\begin{array}{l}
(\circ)(A\ B\ C : \mathsf{Set}) : (\mathsf{El}\ B \to \mathsf{El}\ C) \to (\mathsf{El}\ A \to \mathsf{El}\ B) \to (\mathsf{El}\ A \to \mathsf{El}\ C) \\
f \circ g = \lambda x.\ f\ (g\ x)
\end{array}
$$

$$
\begin{array}{l}
assoc\ (A\ B\ C\ D : \mathsf{Set}) : \\
\quad (f : \mathsf{El}\ C \to \mathsf{El}\ D,\ g : \mathsf{El}\ B \to \mathsf{El}\ C,\ h : \mathsf{El}\ A \to \mathsf{El}\ B) \to \\
\quad \mathsf{Prf}\ (f \circ (g \circ h) == (f \circ g) \circ h)
\end{array}
$$

In general, we write $x\,\Delta : T$ to say that x has type $\Delta \to T$ with Δ implicit. Note that this is a more restricted form of implicit arguments than the one presented in Section 3.6. For every use of x we require that the instantiation of Δ can be inferred using pattern unification [Mil92]. Note that when we have implicit arguments we can replace Id with an infix operator $(==)\ (D : \mathsf{Set}) : \mathsf{El}\ D \to \mathsf{El}\ D \to \mathsf{Prop}$

We conjecture that the conservativity result can be extended to allow the omission of implicit arguments when translating to first-order logic if they

can be inferred from the resulting first-order term. In this case we preserve the property that for a well-typed FOL term there exists a unique typing, which is an important lemma in the conservativity theorem. The kind of implicit arguments we work with can most often be inferred in this way. It is doubtful, however, that it would work for other kinds of implicit arguments such as implicit dictionaries used for overloading.

Omitting the implicit arguments, the formula $f \circ (g \circ h) = (f \circ g) \circ h$ in the context $(A\ B\ C\ D : \mathsf{Set})(f : \mathsf{El}\ C \to \mathsf{El}\ D)(g : \mathsf{El}\ B \to \mathsf{El}\ C)(h : \mathsf{El}\ A \to \mathsf{El}\ B)$ is translated to

$$f \circ (g \circ h) = (f \circ g) \circ h$$

With this translation, the first-order proofs are human readable and, in many cases, correspond closely to a pen and paper proof.

6.4.2 The Plug-in Mechanism

The type checker is equipped with a general plug-in interface that makes it easy to experiment with connections to external tools. A plug-in should implement two functions: a *type checking function* which can be called on particular goals in the program, and a *finalization function* which is called after type checking. A typical usage of these functions is to collect constraints during type checking, and solving the constraints using the external tool at finalization.

To control where the type checking function of a plug-in is invoked we introduce a new form of expressions:

$$\mathsf{Exp} \ ::= \ \ldots \mid \mathit{name}\text{–}\mathbf{plugin}(s_1, \ldots, s_n) \qquad \text{invoking a plug-in}$$

where *name* is the name of a plug-in. It is possible to pass arguments $(s_1, \ldots, s_n)$ to the plug-in. These arguments can be arbitrary expressions which are ignored by the type checker. Hence it is possible to pass ill-typed terms as arguments to a plug-in; it is the responsibility of the plug-in to interpret the arguments. Most plug-ins, of course, expect well-typed arguments and in this case, the plug-in has to invoke the type checker explicitly on its arguments.

6.4.3 The FOL Plug-in

The connection between LF and FOL has been implemented as a plug-in using the mechanism described above. With this implementation we replace the built-in constant () by a call to the plug-in. The idea is that the plug-in

should be responsible for checking the side condition $\Gamma \vdash_{\mathsf{FOL}} P$ in the FOL rule.

An important observation is that decidability of type checking and equality do not depend on the validity of the propositions being checked by the FOL plug-in—nothing will break if the type checker is led to believe that there is an $s : \mathsf{Prf}\bot$. This allows us to delay all first-order reasoning until after type checking. The rationale for doing this is that type checking is cheap and first-order proving is expensive.

Another observation is that it is not feasible to pass the entire context to the prover. Typically, the context contains many things that are not needed for the proof, but would rather overwhelm the prover. To solve this problem, we require that any axioms or lemmas needed to prove a particular goal are passed as arguments to the plug-in. This might seem a severe requirement, but bear in mind that the plug-in is intended for simple goals where you already have an idea of the proof.

More formally, the typing rule for calls to the FOL plug-in is

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash s_1 : \phi_1 \ \ldots \ \Gamma \vdash s_n : \phi_n}{\Gamma \vdash \mathbf{fol{-}plugin}(s_1, \ldots, s_n) : \phi} \ \phi_1, \ldots, \phi_n \vdash_{\mathsf{FOL}} \phi.$$

When faced with a call to a plug-in the type checker calls the type checking function of the plug-in. In this case, the type checking function of the FOL plug-in will verify that the goal is a translatable formula and that the arguments are well-typed proofs of translatable formulæ. If this is the case it will report success to the type checker and store away the side condition in its internal state. After type checking the finalization function of the FOL plug-in is called. For each constraint $\phi_1, \ldots, \phi_n \vdash_{\mathsf{FOL}} \phi$, this function verifies that $[\phi]$ is derivable from $[\phi_1], \ldots, [\phi_n]$ in the resolution calculus by translating the formulæ to clause normal form and feeding them to an external first-order prover (Gandalf, at the moment). If the prover does not manage to find a proof within the given time limit, the plug-in reports an error.

6.5 Examples

The code in this section has been type checked successfully by our prototype type checker. In fact, the typeset version is automatically generated from the actual code. The type checker can infer which types are Sets and which are Props, so we omit El and Prf in the types.

6.5.1 Relational Algebra

Natural numbers can be added to the framework by three new constants $Nat, zero, suc$ plus an axiom for mathematical induction.

$$
\begin{aligned}
&Nat : \mathsf{Set}\\
&zero : Nat\\
&suc : Nat \to Nat\\
&indNat\,(P : Nat \to \mathsf{Prop}) : P\;zero \to ((n : Nat) \to P\;n \to P\,(suc\;n)) \to\\
&\qquad\qquad (m : Nat) \to P\;m
\end{aligned}
$$

Now we fix a set A and consider relations over A. We want to prove that the transitive closure of a symmetric relation is symmetric as well. We define the notion of symmetry and introduce a symbol for relation composition. We could define $R \circ R' = \lambda x \lambda z \exists z.\, x\;R\;y \wedge y\;R'\;z$, but here we only assume that a symmetric relation composed with itself is also symmetric.

$$
\begin{aligned}
&A : \mathsf{Set}\\
&sym : (A \to A \to \mathsf{Prop}) \to \mathsf{Prop}\\
&sym\;R = (x, y : A) \to R\;x\;y \implies R\;y\;x
\end{aligned}
$$

$$
\begin{aligned}
&(\circ) : (A \to A \to \mathsf{Prop}) \to (A \to A \to \mathsf{Prop}) \to (A \to A \to \mathsf{Prop})\\
&axSymO : (R : A \to A \to \mathsf{Prop}) \to sym\;R \to sym\,(R \circ R)
\end{aligned}
$$

We define a monotone chain of approximations $R^{(n)}$ (in the source: $R \hat{}\; n$) of the transitive closure, such that two elements will be related in the transitive closure if they are related in some approximation. The main lemma states that all approximations are symmetric, if R is symmetric.

$$
\begin{aligned}
&(\hat{}) : (A \to A \to \mathsf{Prop}) \to Nat \to (A \to A \to \mathsf{Prop})\\
&axTc : (R : A \to A \to \mathsf{Prop}) \to (x, y : A) \to (n : Nat) \to\\
&\qquad ((R \hat{}\; suc\;n)\;x\;y \Leftrightarrow (R \hat{}\; n)\;x\;y \vee ((R \hat{}\; n) \circ (R \hat{}\; n))\;x\;y)\\
&\quad \wedge\; ((R \hat{}\; zero)\;x\;y \Leftrightarrow R\;x\;y)
\end{aligned}
$$

$$
\begin{aligned}
&main : (R : A \to A \to \mathsf{Prop}) \to sym\;R \to (n : Nat) \to sym\,(R \hat{}\; n)\\
&main\;R\;h = indNat\\
&\qquad\qquad \mathbf{fol{-}plugin}\,(h,\; axTc\;R)\\
&\qquad\qquad (\lambda\, n\; ih \to \mathbf{fol{-}plugin}\,(h,\; axSymO\,(R \hat{}\; n)\;ih,\; axTc\;R,\; ih))
\end{aligned}
$$

Induction is performed at the framework level, base and step case are filled by Gandalf. Pretty printed, Gandalf produces the following proof of

the step case:

(1)	$\forall xy.$	$(R^{(n)} \circ R^{(n)})\, x\, y \Longrightarrow (R^{(n)} \circ R^{(n)})\, y\, x$	
(2)	$\forall mxy.$	$R^{(suc\, m)}\, x\, y \Longrightarrow (R^{(m)} \circ R^{(m)})\, x\, y \vee R^{(m)}\, x\, y$	
(3)	$\forall mxy.$	$(R^{(m)} \circ R^{(m)})\, x\, y \Longrightarrow R^{(suc\, m)}\, x\, y$	
(4)	$\forall mxy.$	$R^{(m)}\, x\, y \Longrightarrow R^{(suc\, m)}\, x\, y$	
(5)	$\forall xy.$	$R^{(n)}\, x\, y \Longrightarrow R^{(n)}\, y\, x$	
(6)		$R^{(suc\, n)}\, a\, b$	
(7)		$R^{(suc\, n)}\, b\, a \Longrightarrow \bot$	
(8)		$(R^{(n)} \circ R^{(n)})\, a\, b \vee R^{(n)}\, a\, b$	(2), (6)
(9)		$(R^{(n)} \circ R^{(n)})\, b\, a \vee R^{(n)}\, a\, b$	(1), (8)
(10)		$R^{(n)}\, a\, b$	(3), (7), (9)
(11)		$R^{(n)}\, b\, a$	(5), (10)
(12)		$\bot$	(4), (7), (11)

The transitive closure is now defined as $TC\, R\, x\, y = \exists n.\, R^{(n)} xy$. To formalize this, we add existential quantification and its proof rules. The final theorem demostrates how existential quantification can be handled in the framework.

$$
\begin{array}{l}
\mathit{Exists}\,(X : \mathsf{Set}) : (X \to \mathsf{Prop}) \to \mathsf{Prop} \\
\mathit{existsI}\,(X : \mathsf{Set})(P : X \to \mathsf{Prop}) : (x : X) \to P\, x \to \mathit{Exists}\, P \\
\mathit{existsE}\,(X : \mathsf{Set})(P : X \to \mathsf{Prop})(C : \mathsf{Prop}) : \\
\quad \mathit{Exists}\, P \to ((x : X) \to P\, x \to C) \to C
\end{array}
$$

$$
\begin{array}{l}
TC : (A \to A \to \mathsf{Prop}) \to A \to A \to \mathsf{Prop} \\
TC\, R\, x\, y = \mathit{Exists}\,(\lambda\, n \to (R\, \hat{}\, n)\, x\, y)
\end{array}
$$

$$
\begin{array}{l}
\mathit{thm} : (R : A \to A \to \mathsf{Prop}) \to \mathit{sym}\, R \to \mathit{sym}\,(TC\, R) \\
\mathit{thm}\, R\, h\, x\, y = \mathit{impI}\,(\lambda\, p \to \\
\qquad\qquad \mathit{existsE}\, p\,(\lambda\, n\, q \to \mathit{existsI}\, n\, \mathbf{fol{-}plugin}(q,\ \mathit{main}\, R\, h\, n)))
\end{array}
$$

6.5.2 Category Theory

One application of the FOL plug-in is to category theory. Typically, proofs in category theory contain a fair amount of symbolic manipulation, something which we can leave to the plug-in.

To reason about category theory we introduce the appropriate constants together with their axioms.

$$
\mathit{Obj} : \mathsf{Set}
$$

$$Hom : Obj \to Obj \to \mathsf{Set}$$

$$id\,(a : Obj) : Hom\ a\ a$$
$$(\circ)\,(a,\ b,\ c : Obj) : Hom\ b\ c \to Hom\ a\ b \to Hom\ a\ c$$

$$axId_1\,(a,\ b : Obj) : (f : Hom\ a\ b) \to f\ ==\ id \circ f$$
$$axId_2\,(a,\ b : Obj) : (f : Hom\ a\ b) \to f\ ==\ f \circ id$$

$$assoc\,(a,\ b,\ c,\ d : Obj) :$$
$$\quad (f : Hom\ c\ d) \to (g : Hom\ b\ c) \to (h : Hom\ a\ b) \to$$
$$\quad (f \circ g) \circ h\ ==\ f \circ (g \circ h)$$

Now we can define what it means for a morphism to be *epi* and prove that if the composition of two morphisms is epi then the first morphism must also be epi.

$$isEpi\,(a,\ b : Obj) : Hom\ a\ b \to \mathsf{Prop}$$
$$isEpi\,\{\_\}\,\{b\}\,f = (c : Obj) \to (g,\ h : Hom\ b\ c) \to$$
$$\qquad\qquad g \circ f\ ==\ h \circ f \implies g\ ==\ h$$

$$prop\,(a,\ b,\ c : Obj) : (f : Hom\ b\ c) \to (k : Hom\ a\ b) \to$$
$$\qquad\qquad isEpi\,(f \circ k) \implies isEpi\,f$$
$$prop\,f\,k = impI\,(\lambda epi\_kf \to \mathbf{fol{-}plugin}(assoc,\ epi\_kf))$$

Gandalf has no problem proving this (very simple) proposition and, more importantly, the proof that Gandalf produces is very close to the proof we would write by hand. Pretty printed, the proof we get looks as follows.

(1)	$\forall X\,Y\,Z.$	$(X \circ Y) \circ Z = X \circ (Y \circ Z)$	
(2)	$\forall X\,Y.$	$X \circ (f \circ k) = Y \circ (f \circ k) \implies X = Y$	
(3)		$g \circ f == h \circ f$	
(4)		$g == h \implies \bot$	
(5)	$\forall X.$	$g \circ (f \circ X) == h \circ (f \circ X)$	$\{(1), (3)\}$
(6)		$\bot$	$\{(2), (4), (5)\}$

6.5.3 Computer Algebra

An example from M. Beeson [Bee07]. This example illustrates how we can combine the interactive style of the logical framework, for instance for the induction steps, with the first-order logic plugin.

In this example we want to reason about existentially quantified propositions so we add some new constants to the signature.

$$
\begin{aligned}
&\mathit{Exists}\,(A : \mathsf{Set}) : (A \to \mathsf{Prop}) \to \mathsf{Prop}\\
&\mathit{existsI}\,(A : \mathsf{Set}) : (P : A \to \mathsf{Prop}) \to (x : A) \to P\,x \to \mathit{Exists}\,P\\
&\mathit{existsE}\,(A : \mathsf{Set}) :(P : A \to \mathsf{Prop}) \to \mathit{Exists}\,P \to\\
&\qquad\qquad (C : \mathsf{Prop}) \to ((x : A) \to P\,x \implies C) \to C
\end{aligned}
$$

We also need natural numbers. For this use the datatype extensions which allows us to define recursive functions over the natural numbers. For instance, we can write a recursive proof of the induction principle.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Nat} : \mathsf{Set}\ \textbf{where}\\
&\ \mathsf{zero} : \mathit{Nat}\\
&\ \mathsf{suc} : \mathit{Nat} \to \mathit{Nat}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{indNat} : (P : \mathit{Nat} \to \mathsf{Prop}) \to P\,\mathsf{zero} \to\\
&\qquad ((n : \mathit{Nat}) \to P\,n \implies P\,(\mathsf{suc}\,n)) \to\\
&\qquad (x : \mathit{Nat}) \to P\,x\\
&\mathit{indNat}\,P\,a\,g\,\mathsf{zero} = a\\
&\mathit{indNat}\,P\,a\,g\,(\mathsf{suc}\,n) = \mathit{impE}\,(g\,n)\,(\mathit{indNat}\,P\,a\,g\,n)
\end{aligned}
$$

The goal of the example is to prove that in an integral ring, the only nilpotent element is zero. We start by defining what it means to be an integral ring.

$$
\begin{aligned}
&\mathit{isRing} : (R : \mathsf{Set}) \to (R \to R \to R) \to (R \to R \to R) \to\\
&\qquad\qquad (R \to R) \to R \to R \to \mathsf{Prop}\\
&\mathit{isRing}\,R\,(+)\,(*)\,\mathit{minus}\,\mathit{Zero}\,\mathit{One} =\\
&\ (x : R) \to (y : R) \to (z : R) \to\\
&\ (\,(x + y) == (y + x)\\
&\quad \wedge\ (x + \mathit{Zero}) == x\\
&\quad \wedge\ (x + (\mathit{minus}\,x)) == \mathit{Zero}\\
&\quad \wedge\ (x + (y + z)) == ((x + y) + z)\\
&\quad \wedge\ (x * (y + z)) == ((x * y) + (x * z))\\
&\quad \wedge\ ((y + z) * x) == ((y * x) + (z * x))\\
&\quad \wedge\ (x * \mathit{One}) == x\\
&\quad \wedge\ (\mathit{One} * x) == x\\
&\quad \wedge\ (x * (y * z)) == ((x * y) * z)\\
&\ )
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{isIntegral} : (R : \mathsf{Set}) \to (R \to R \to R) \to R \to \mathsf{Prop}\\
&\mathit{isIntegral}\,R\,(*)\,\mathit{Zero} =\\
&\qquad (x : R) \to (y : R) \to x * y == \mathit{Zero} \implies
\end{aligned}
$$

$$x == Zero \vee y == Zero$$

In the following we work on a particular (but abstract) integral ring.

```
R : Set
( + ) : R → R → R
( * ) : R → R → R
minus : R → R
Zero : R
One : R

axR : isRing R ( + ) ( * ) minus Zero One
axI : isIntegral R ( * ) Zero

power : Nat → R → R
power zero x = One
power (suc n) x = (power n x) * x

isZero : R → Prop
isZero x = x == Zero

isNilpotent : R → Prop
isNilpotent x = Exists (λ n → isZero (power n x))
```

This is all we need to start the proof. First we prove some lemmas.

```
lemCancel : (x : R) → (y : R) → x + y == y ⟹ isZero x
lemCancel x y =
  impI (λh →
          let rem : isZero (x + (y + minus y))
              rem = fol−plugin(h, axR)
          in
            fol−plugin(rem, axR)
        )
```

The proof of $Zero * x == Zero$ is not trivial (but can be done purely automatically if desired) so we give the main steps of one possible proof explicitly.

```
lemZero : (x : R) → isZero (Zero * x)
lemZero x =
```

```
  let rem₁ : Zero + One == One
      rem₁ = fol−plugin(axR)
      rem₂ : (Zero + One) * x == Zero * x + One * x
      rem₂ = fol−plugin(axR)
      rem₃ : Zero * x + One * x == One * x
      rem₃ = fol−plugin(axR, rem₁, rem₂)
  in
    fol−plugin(rem₃, lemCancel)

lemOneZero : (x : R) → One == Zero ⟹ isZero x
lemOneZero x = fol−plugin(axR, lemZero)
```

The main lemma is proved by induction explicitly at the framework level.

```
prop : R → Nat → Prop
prop x n = isZero (power n x) ⟹ isZero x

lemMain : (x : R) → (n : Nat) → prop x n
lemMain x =
  let base : prop x zero
      base = fol−plugin(lemOneZero)
      step : (n : Nat) → prop x n ⟹ prop x (suc n)
      step n = fol−plugin(axR, axI)
  in
    indNat (prop x) base step

thm : (x : R) → isNilpotent x → isZero x
thm x h = existsE (λn → isZero (power n x)) h (isZero x) (lemMain x)
```

6.6 Related Work

Smith and Tammet [ST95] also combine Martin-Löf type theory and first-order logic, which was the original motivation for creating the system Gandalf. The main difference to their work is that we use implicit typing and restrict to quantifier-free formulæ. An advantage is that we have a simple translation, and hence get a quite direct connection to resolution theorem provers. Hence, we can hope, and this has been tested positively in several examples, that the proof traces we get from the prover are readable as such and therefore can been used as a proof certificate or as feedback for the user. For instance, the user can formulate new lemmas suggested by this

proof trace. We think that this aspect of readability is more important than creating an explicit proof term in type theory (which would actually be less readable). It should be stressed that our conservativity result contains, since it is constructive, an algorithm that can transform the resolution proof to a proof in type theory, if this is needed.

Huang et. al. [HKK+94] present the design of Ω-MKRP[6], a tool for the working mathematician based on higher-order classical logic, with a facility of proof planning, access to a mathematical database of theorems and proof tactics (called methods), and a connection to first-order automated provers. Their article is a well-written motivation for the integration of human and machine reasoning, where they envision a similar division of labor as we have implemented. We have, however, not addressed the problem of mathematical knowledge management and proof tactics.

Wick and McCune [WM89] list three options for connecting type systems and FOL: include type literals, put type functions around terms, or use implicit typing. We rediscovered the technique of implicit typing and found out later that it is present already in the work of Beeson [Bee07]. Our work shows that this can also be used with dependent types, which is not obvious a priori. Our formulation of the correctness properties, as a conservativity statement, requires some care (with the role of the sort Prop), and is an original contribution.

Bezem, Hendriks, and de Nivelle [BHdN02] describe how to transform a resolution proof to a proof term for *any* first-order formula. However, the resulting proof terms are hard to read for a human because of the use of skolemisation and reduction to clausal forms. Furthermore, they restrict to a fixed first-order domain.

Hurd's work on a Gandalf-tactic for HOL [Hur99] is along the same lines. He translates untyped first-order HOL goals to clause form, sends them to Gandalf and constructs an LCF proof from the Gandalf output. In later work [Hur02, Hur03] he handles types by having two translations: the untyped translation, and a translation with explicit types. The typed translation is only used when the untyped translation results in an ill-typed proof.

JProver [SLKN01] is a connection-based intuitionistic theorem prover which produces proof objects. It has been integrated into NuPrl and Coq. The translation from type theory to first-order logic involves some heuristics when to include or discard type information. Unfortunately, the description [SLKN01] does not contain formal systems or correctness arguments, but focuses on the connection technology.

Jia Meng and Paulson [MP04] have carried out substantial experiments

[6]Markgraf Karl Refutation Procedure.

on how to integrate the resolution theorem prover Vampire into the interactive proof tool Isabelle. Their translation from higher-order logic (HOL) to first-order logic keeps type information, since HOL supports overloading via axiomatic type classes and discarding type information for overloaded symbols would lead to unsound reasoning. They claim to cut down the search space via type information, but this is also connected to overloading. The aim of their work is different to ours: while they use first-order provers to do as much automatic proofs and proof search as possible, we employ automation only to liberate the user from seemingly trivial proof steps.

In Coq, NuPrl, and Isabelle, the user constructs a proof via tactics. We provide type theory as a proof language in which the user writes down a proof skeleton, consisting of lemmas, scoped hypotheses, invocation of induction, and major proof steps. The first-order prover is invoked to solve (easy) subgoals. This way, we hope to obtain human-readable proof documents (see our examples).

6.7 Future Work

The logical framework used in this chapter does not support Σ-types. However, the extension of the translation to FOL is straightforward, we simply add a new binary function symbols for representing pairs. A more substantial extension is the addition of datatypes and functions defined by pattern matching. With this extension, it is possible to represent each connective as a parameterized data type. Each introduction rule is represented by a constructor, and the elimination rules are represented by functions defined by cases. This gives a computational justification of each of the axioms of the signature Σ_{nat}. The extension of the translation to FOL is also straightforward: each defined equations for functions becomes a FOL equality. One needs also to express that each constructor is one-to-one and that terms with distinct constructors are distinct.

Another direction of further work is to extend the conservativity theorem to handle implicit arguments. We also think that it is possible to extend our class of translatable formulæ, for instance, to include some cases of existential quantification.

One could think of adding more plug-ins, with the same principle that they are justified by a general metatheorem. For instance, one could add a plug-in to a model checker, or a plug-in to a system with a decision procedure for Presburger arithmetic.

A different approach, which is some ways is more appealing, is to implement certified provers internally in the language, in the way that was done for

equations in commutative monoids in Section 5.2. However, implementing an efficient FOL prover in a dependently type language is no small challenge.

Acknowledgments. We thank the members of the Cover project, especially Koen Claessen for discussions on implicit typing and the clausification tool Santa for a uniform connection to FOL provers, and Grégoire Hamon for programming the clausifier of the FOL plug-in in a previous version.

Chapter 7

Conclusions

The main goal of this thesis has been to pave the way for practical programming languages with dependent types. In pursuing this goal we have looked at a number of topics: pattern matching (Chapter 2), metavariables (Chapter 3), module systems (Chapter 4), and automation (Chapter 6). Furthermore we have designed and implemented a programming language, Agda, show-casing our results (Chapter 5).

Pattern matching

Dependent types and in particular inductive families of types brings new dimensions to pattern matching not present in the simply typed case. This was observed by Coquand who outlined an algorithm for incrementally constructing functions defined by pattern matching [Coq92], which was consequentially implemented in ALF [MN94]. Later McBride [McB99, MM04a, GMM06] showed how pattern match definitions can be reduced to definitions by elimination rules given uniqueness of identity proofs. In this thesis we have given a direct type checking algorithm for pattern match equations supporting the *with* rule [MM04a]. Our algorithm is more liberal than previous approaches in that it allows overlapping pattern equations.

Metavariables and implicit syntax

When working in a monomorphic type theory, the ability to omit the parts of the program that can be inferred automatically becomes an important feature. This not only makes programs easier to read, but also improves the performance of the type checker [NL98]. To do this one typically inserts metavariables for the omitted terms. The type checker will then attempt to infer the values of these metavariables.

We have given a type checking algorithm for a dependently typed logic extended with metavariables. To maintain the important invariant that terms being evaluated are type correct we work with well-typed approximations of terms, where potentially ill-typed subterms have been replaced by constants. We showed that type checking is decidable and that the algorithm is sound.

We presented the type checking algorithm for a simple dependently typed logical framework **MLF**, but outlined how it can be extended to more feature-rich logics. The implementation handles the full logic of Agda, and has proven to work well with examples of several thousand metavariables.

Module system

In larger developments it is crucial to be able to split a program into separate units, and to manage the scope of these units so that definitions from one unit is not automatically visible in all others. For this purpose, we have presented a reasonable simple and easy to implement module system which is still expressive enough to allow large programs to be structured in a nice way. A key design decision was to keep the module system and the type system as separate as possible. As a result the module system is largely independent of the underlying language.

Automation

When working with the more precise types that a dependently typed language enables, it is sometimes necessary or desirable to prove properties of your programs. While these can be constructed directly in the type theory, this is sometimes tedious work. To alleviate proving simple first-order properties, we described the implementation of a logical framework with proof-irrelevant propositions and its connection to the automatic first-order logic prover Gandalf. Soundness and conservativity of the connection was established by general metatheorems. By restricting the set of formulas under consideration to that of *geometric* formulas we obtained a simple, transparent translation between the framework and first-order logic. Moreover the proofs constructed by the prover are intuitionistically valid.

Agda

We have collected the features described in this thesis[1] in a language Agda. While it is still far from being a fully fledged programming language, it

[1] With the exception of the first-order logic connection, which has been implemented in the AgdaLight language [Nor06]

has still managed to gather a handful of users who have written some quite impressive programs [AC07, AMS07, BD07, Dan06, Dan07, SA07].

Future work

Though the work in this thesis has taken us closer to our goal of a practical dependently typed programming language, there are many areas we have left unexplored. Some of these have been explored by others whereas some remain open problems.

One topic which we have not touched upon, but which is crucial for a programming language, is program compilation. Dependent types offer some exciting possibilities for type directed optimizations not available in a simply typed language. This was explored by Brady [Bra05] with promising results.

Another topic of interest is effectful programming. In a dependently typed language where computation happens at compile time it is important to distinguish pure computations and effectful computations—we do not want any effects to happen during type checking. On the other hand we would like to be able to reason about effectful computations at compile time. One way of achieving this is to build a model of the effects inside the language which is used for type checking, but switch to the real thing when executing. How to build such a model has been studied [HS00, SA07], but there is a lot work still to be done.

An interesting, but perhaps not crucial, topic is that of multi-staged programming and reflection. We touched upon this in Section 5.2 where we observed that reflection would enable us to create nicer interfaces to internal tactics, such as the presented prover for equations in a commutative monoid. Some interesting work has been done in this direction by Brady and Hammond [BH06].

Perhaps the most important challenge we are now facing is that of learning to program with dependent types. This topic was pioneered by McBride and McKinna [MM04a], but unfortunately there has been no programming language in which to put their ideas to the test on a larger scale. Agda might not be that language just yet, but it is a good step along the way.

Bibliography

[AC05] Andreas Abel and Thierry Coquand. Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. In Paweł Urzyczyn, editor, *TLCA'05*, volume 3461 of *LNCS*, pages 23–38. Springer, April 2005.

[AC07] Thorsten Altenkirch and James Chapman. Big step normalisation. In submission, 2007.

[ACN05] Andreas Abel, Thierry Coquand, and Ulf Norell. Connecting a logical framework to a first-order logic prover. In B. Gramlich, editor, *Proceedings of 5th International Workshop on Frontiers of Combining Systems, Lecture Notes in Artificial Intelligence*, volume 3717, pages 285–301. Springer-Verlag, September 2005.

[ACT07] Andrea Asperti, Claudio Sacerdoti Coen, and Enrico Tassi. Types summer school, 2007. `http://typessummerschool07.cs.unibo.it/`.

[AMM05] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV'07: Proceedings of the Programming Languages meets Program Verification Workshop*, 2007.

[Aug85] L. Augustsson. Compiling Pattern Matching. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 368–381, Nancy, France, 1985.

[Aug98] Lennart Augustsson. Cayenne — a language with dependent types. In *Proc. of the International Conference on Functional Programming (ICFP'98)*. ACM Press, September 1998.

[Bar92a] H. P. Barendregt. Typed lambda calculi. In S. Abramsky et al., editor, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[Bar92b] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, volume 2. 1992.

[BC03] M. Bezem and T. Coquand. Newman's lemma—a case study in proof automation and geometric logic. *Bull. Eur. Assoc. Theor. Comput. Sci. EATCS No. 79*, pages 86–100, 2003.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BD07] Alexandre Buisse and Peter Dybjer. Towards formalizing categorical models of type theory in type theory. In Brigite Pientka and Carsten Schurmann, editors, *Second International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP'07)*, Electronic Notes in Theoretical Computer Science, pages 72–85. Elsevier, 2007.

[Bee07] Michael Beeson. Otter-λ home page, 2007. `http://michaelbeeson.com/research/otter-lambda`.

[BH06] Edwin Brady and Kevin Hammond. A verified staged interpreter is a verified compiler: Multi-stage programming with dependent types. In *Proc. Conf. Generative Programming and Component Engineering (GPCE '06), Portland, Oregon*, Lecture Notes in Computer Science. Springer, 2006. To appear.

[BHdN02] Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. *JAR*, 29(3–4):253–275, 2002. Special Issue *Mechanizing and Automating Mathematics: In honour of N.G. de Bruijn*.

[Bra05] Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.

[Bru72] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34(5):381–392, 1972.

[CAB+86] 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 Development System*. Prentice-Hall, NJ, 1986.

[CC99] Catarina Coquand and Thierry Coquand. Structured type theory. In *Workshop on Logical Frameworks and Meta-languages (LFM'99)*, Paris, France, Sep 1999.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming*, pages 268–279. ACM, 2000.

[Chl06] Adam Chlipala. Modular development of certified program verifiers with a proof assistant. In *Proceedings of 11th ACM SIGPLAN International Conference on Functional Programming (ICFP'06)*, September 2006.

[Chl07] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*, June 2007.

[Chr03] Jacek Chrząszcz. Implementation of modules in the Coq system. In David Basin and Burkhart Wolff, editors, *Proceedings of the Theorem Proving in Higher Order Logics 16th International Conference*, volume 2758 of *LNCS*, pages 270–286, Rome, Italy, September 2003. Springer.

[CLR01] Michel Coste, Henri Lombardi, and Marie-Françoise Roy. Dynamical methods in algebra: Effective Nullstellensätze. *APAL*, 111(3):203–256, 2001.

[Coq92] T. Coquand. Pattern matching with dependent types. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

[Coq96] T. Coquand. An algorithm for type-checking dependent types. *Comput. Programming 26*, pages 167–177, January 1996.

[Cou07] Judicaël Courant. MC_2 A module calculus for Pure Type Systems. *Journal of Functional Programming*, 17:287–352, 2007.

[CPT] T. Coquand, R. Pollack, and M. Takeyama. A logical framework with dependently typed records. In *Typed lambda calculi and applications (2003), Lecture Notes in Comput. Sci., 2701*, pages 22–28.

[Dan06] Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In *TYPES 2006*. Springer-Verlag, 2006.

[Dan07] Nils Anders Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. Draft, 2007.

[dB80] Niklas G. de Bruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in combinatory logic, lambda calculus and formalism*, pages 579–606, London-New York, 1980. Academic Press.

[dB91a] N. G. de Bruijn. A plea for weaker frameworks. pages 40–67, 1991.

[dB91b] N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91(2):189–204, 1991.

[DHK95] Gilles Dowek, Therese Hardin, and Claude Kirchner. Higher-order unification via explicit substitutions. In Dexter Kozen, editor, *Proceedings of the Tenth Annual IEEE Symp. on Logic in Computer Science, LICS 1995*, pages 366–374. IEEE Computer Society Press, June 1995.

[Dow01] Gilles Dowek. Higher-order unification and matching. *Handbook of automated reasoning*, pages 1009–1062, 2001.

[DS06] Peter Dybjer and Anton Setzer. Indexed induction-recursion. *The Journal of Logic and Algebraic Programming*, 66(1):1–49, January 2006.

[Dyb94] P. Dybjer. Inductive families. *Formal Aspects of Computing*, pages 440–465, 1994.

[Ell89] C. M. Elliot. Higher-order unification with dependent function types. In N. Derikowitz, editor, *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, pages 121–136, April 1989.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935.

[GMM06] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Goguen Festschrift*, volume 4060 of *LNCS*. Springer Verlag, 2006.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *JACM*, 40(1):143–184, 1993.

[HKK+94] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jörn Richts, and Jörg H. Siekmann. Omega-MKRP: A proof development environment. In Alan Bundy, editor, *CADE'94*, volume 814 of *LNCS*, pages 788–792. Springer, 1994.

[HP98] Robert Harper and Frank Pfenning. A module system for a programming language based on the lf logical framework. *Journal of Logic and Computation*, 8(1):5–31, 1998.

[HS94] Martin Hofmann and Thomas Streicher. A groupoid model refutes uniqueness of identity proofs. In *LICS 1994*, pages 208–212. IEEE Press, 1994.

[HS00] Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic, 14th international workshop*, volume 1862 of *Springer Lecture Notes in Computer Science*, pages 317–331. Springer-Verlag, 2000.

[Hue75] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

[Hur99] Joe Hurd. Integrating Gandalf and HOL. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *TPHOLS'99*, volume 1690 of *LNCS*, pages 311–321. Springer, September 1999.

[Hur02] Joe Hurd. An LCF-style interface between HOL and first-order logic. In Andrei Voronkov, editor, *CADE'02*, volume 2392 of *LNAI*, pages 134–138. Springer, 2002.

[Hur03] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *STRATA'03*, number CP-2003-212448 in NASA Technical Reports, pages 56–68, September 2003.

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In *FPCA*, pages 190–203, 1985.

[Lam93] Leslie Lamport. How to write a proof. In *Global Analysis in Modern Mathematics*, pages 311–321. Publish or Perish, Houston, Texas, U.S.A., February 1993. Also appeared as SRC Research Report 94.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[Luo94] Zhaohui Luo. *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., New York, NY, USA, 1994.

[McB99] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[McB06] Conor McBride, 2006. Personal communication.

[McB07] Conor McBride. Epigram, 2007. `http://www.e-pig.org`.

[Mil91] D. Miller. Unification of simply typed lambda-terms as logic programming. In K. Furukawa, editor, *Logic Programming: Proc. of the Eighth International Conference*, pages 255–269. MIT Press, Cambridge, MA, 1991.

[Mil92] Dale Miller. Unification under a mixed prefix. *J. Symb. Comput.*, 14(4):321–358, 1992.

[Miq01] Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In S. Abramsky, editor, *Proc. of 5th Int. Conf.*

on Typed Lambda Calculi and Applications, TLCA'01, Krakow, Poland, 2–5 May 2001, volume 2044, pages 344–359. Springer-Verlag, Berlin, 2001.

[ML72] P. Martin-Löf. An Intuitionistic Theory of Types. Technical report, University of Stockholm, 1972.

[ML75] P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium 1973*, pages 73–118, Amsterdam, 1975. North-Holland Publishing Company.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[MM04a] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, January 2004.

[MM04b] Conor McBride and James McKinna. I am not a number; I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN Haskell Workshop.* ACM Press, 2004.

[MN94] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[MP04] Jia Meng and Lawrence C. Paulson. Experiments on supporting interactive proof using resolution. In David A. Basin and Michaël Rusinowitch, editors, *IJCAR'04*, volume 3097 of *LNCS*, pages 372–384. Springer, 2004.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Muñ01] César Muñoz. Proof-term synthesis on dependent-type systems via explicit substitutions. *Theor. Comput. Sci.*, 266(1-2):407–440, 2001.

[NC07] Ulf Norell and Catarina Coquand. Type checking in the presence of metavariables. Unpublished, 2007.

[Nip93] Tobias Nipkow. Functional unification of higher-order patterns. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 64–74, 1993.

[NL98] G. Necula and P. Lee. Efficient representation and validation of proofs. In *LICS'98*, pages 93–104. IEEE, June 1998.

[Nor06] Ulf Norell. Agda light, 2006. `http://www.cs.chalmers.se/~ulfn/agdaLight`.

[Nor07] Ulf Norell. Agda 2, 2007. `http://www.cs.chalmers.se/~ulfn/Agda`.

[NPP07] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 2007. To appear.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[NPS00] Bengt Nordström, Kent Petersson, and Jan Smith. Martin-Löf's type theory. In *Handbook of Logic in Computer Science*, volume 5. OUP, October 2000.

[Pau90] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361 – 386. Academic Press, 1990.

[Pfe91] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, 1991.

[PHe+99] S. Peyton Jones, J. Hughes, (editors), L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.

[Pol90] R. Pollack. Implicit syntax. In the preliminary Proceedings of the 1st Workshop on Logical Frameworks, 1990.

[Pol94] R. Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1994.

[Pol00] Robert Pollack. Dependently typed records for representing mathematical structure. `www.dcs.ed.ac.uk/~rap/export/records.ps`, 2000.

[PPM90] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 209–228. Springer-Verlag, 1990.

[PS07] Adam Poswolsky and Carsten Schürmann. Delphin: A functional programming language with higher-order encodings and dependent types. In submission, 2007.

[PVWW06] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, Portland, Oregon, September 2006. ACM SIGPLAN.

[Pym90] D. Pym. *Proof, search and computation in general logic*. PhD thesis, University of Edinburgh, 1990.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, January 1965.

[Ré93] Didier Rémy. Syntactic theories and the algebra of record terms, 1993.

[SA07] Wouter Swierstra and Thorsten Altenkirch. Beauty in the beast: A functional semantics of the awkward squad. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell*, 2007.

[She05] Tim Sheard. Putting curry-howard to work. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85, New York, NY, USA, 2005. ACM Press.

[SLKN01] Stephan Schmitt, Lori Lorigo, Christoph Kreitz, and Aleksey Nogin. JProver: Integrating connection-based theorem proving into interactive proof assistants. In R. Gore, A. Leitsch, and T. Nipkow, editors, *IJCAR'01*, volume 2083 of *LNAI*, pages 421–426. Springer, 2001.

[Soz07] Matthieu Sozeau. Subset coercions in Coq. In *TYPES'06*, volume 4502 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2007.

[SP03] C. Schürmann and F. Pfenning. A coverage checking algorithm for lf. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics, TPHOLs*, 2003.

[ST95] Jan M. Smith and Tanel Tammet. Optimized encodings of fragments of type theory in first-order logic. In Stefano Berardi and Mario Coppo, editors, *TYPES'95*, volume 1158 of *LNCS*, pages 265–287. Springer, 1995.

[Str93] Thomas Streicher. Investigations into intensional type theory. Habilitation Thesis, Ludwig Maximilian Universität, 1993.

[Tam97] Tanel Tammet. Gandalf. *JAR*, 18(2):199–204, 1997.

[WM89] C. A. Wick and W. McCune. Automated reasoning about elementary point-set topology. *Journal of Automated Reasoning*, 5(2):239–255, 1989.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1998.

[Xi04] Hongwei Xi. Applied Type System (extended abstract). In *post-workshop Proceedings of TYPES 2003*, pages 394–408. Springer-Verlag LNCS 3085, 2004.