Reflection in Agda

Paul van der Walt

M.Sc. thesis ICA-3120805
[*Supervisors*] Wouter Swierstra and Johan Jeuring

rev. 58b3ae3, October 4, 2012

Universiteit Utrecht

Department of Computing Science

Abstract

This project explores the recent addition to Agda enabling *reflection*, in the style of Lisp, MetaML, and Template Haskell. It illustrates several possible applications of reflection that arise in dependently typed programming, and details the limitations of the current implementation of reflection. Examples of type-safe metaprograms are given that illustrate the power of reflection coupled with a dependently typed language. Among other things the limitations inherent in having `quote` and `unquote` implemented as keywords are highlighted. The fact that lambda terms are returned without typing information is discussed, and a solution is presented. Also provided is a detailed users' guide to the reflection API and a library of working code examples to illustrate how various common tasks can be performed, along with suggestions for an updated reflection API in a future version of Agda.

"Using Coq is like doing brain surgery over the telephone."
– Peter Hancock

Contents

Chapter 1

Introduction

Since the inception of computer programming, one of the aims has been to write code as concisely as possible, while achieving the most powerful effect. One of the holy grails of writing programs is also being able to reuse pieces of code, after having written them once, as opposed to continually writing small variations on existing code. Reinventing the wheel is something the programmer should not relish doing.

One of the many techniques for writing more effective code is that of *metaprogramming*, which refers to the ability of a program to inspect[1] its own code and modify it. To the uninitiated, this sounds rather magical [54], but it has long been a favourite feature of users of such languages as Lisp [46]. In many cases, this allows code to be a lot more concise and general, and thus reusable, than usually is possible in simple languages.

The dependently typed programming language Agda [42, 43] has recently been extended with a *reflection mechanism* for compile time metaprogramming in the style of Lisp [46], MetaML [56], Template Haskell [51], and C++ templates [3]. Agda's reflection mechanisms make it possible to convert a program fragment into its corresponding abstract syntax tree and vice versa. In tandem with Agda's dependent types, this has promising new programming potential.

The main questions we aim to answer during this project are:

> "What are interesting applications of the new reflection API? Which tedious tasks can we automate? What advantages does the combination of dependent types and reflection give us? Finally, is the reflection API adequate as it stands to facilitate our needs or does it require extension? If extension is necessary, what kind and how much?"

This project starts to explore the possibilities and limitations of this new reflection mechanism. It describes several case studies, exemplative of the kind of problems that can be solved using reflection. More specifically it makes the following contributions:

- A short *introduction to Agda* as a programming language is given in Chapter 2.

[1] or *reflect* upon

- The current status of the reflection mechanism is documented. The existing documentation is limited to a paragraph in the release notes [1] and comments in the compiler's source code. In Chapter 3 we give several short examples of *the reflection API*[2] *in action*.
- How to use Agda's reflection mechanism to automate certain categories of proofs is illustrated in Chapter 4. The idea of *proof by reflection* is certainly not new, but still worth examining in the context of this new technology.
- We show how to write *type-safe metaprograms*. To illustrate this point, we will develop a type-safe translation from the simply typed lambda calculus to programs in continuation-passing style (CPS), followed by a type-safe translation of closed lambda terms into SKI combinator calculus (Chapter 5). In doing this, structurally recursive, total, type preserving CPS and SKI transformations are defined.
- Finally, we also discuss some of the *limitations of the current implementation* of reflection (Chapter 6), brought to light by attempts to automate certain aspects of generic programming.

The code and examples presented in this paper all compile using the latest development version of Agda (currently 2.3.1), with some minor modifications to the compiler (see Appendix A.1). All code, including this report, is available on GitHub[3]. This thesis is also a Literate Agda file, which means the code snippets can be extracted, compiled and played around with.

[2] API stands for *application programming interface*. The reflection API is an interface to Agda's internal representation of terms.

[3] https://github.com/toothbrush/reflection-proofs

Chapter 2

Introducing Agda

Besides being a common Swedish female name and alluding to a certain hen[1] in Swedish pop culture[2], Agda is an implementation of Martin-Löf's type theory [35], extended with records and modules. Agda is developed at the Chalmers University of Technology [42]; thanks to the Curry–Howard isomorphism, it is both a functional[3] functional[4] programming language and a proof assistant for intuitionistic logic. It is comparable with Coquand's calculus of constructions, the logic behind Coq [13]. Coq is similarly both a programming language and proof assistant.

In informal terms, the Curry–Howard isomorphism states that there is a correspondence between types and propositions on the one hand, and programs and proofs on the other hand [53]. The interpretation of a programming language as a logic is that types express theorems which can be proven by providing an implementation. This correspondence is outlined further in Sec. 2.3.

In Agda, types of functions are allowed to *depend upon* values – the main difference between a dependently typed programming language and a simply typed language is that the divide between the world of values and that of types is torn down.

This chapter aims to provide a crash course on Agda. The reader is presumed to be fluent in GHC Haskell; the fact that Agda's syntax is inspired by Haskell makes it a reasonable choice to explain most of the concepts here from a Haskell programmer's point of view. Consequently, users familiar with programming in Haskell should be able to hit the ground running in Agda.

2.1 First Steps in Agda

Our short tutorial starts slowly; we will look at how textbooks define natural numbers, in so-called Peano style. A single colon means "is of type", so in Fig. 2.1, zero is of type Natural. The constructor suc has type Natural → Natural, which means that it takes a natural as argument and produces a new natural.

[1] . . . bearing in mind that *coq* means rooster in French. . .

[2] See Cornelis Vreeswijk's song about Agda, a hen, at `http://youtu.be/zPY42kkRADc`.

[3] Functional as in practically usable.

[4] Functional as in Haskell.

This new natural is also the successor of the old natural. This inductive style of data type definitions is a frequently used technique in both Haskell and Agda.

```
data Natural : Set where
  zero :              Natural
  succ : Natural → Natural
```

Figure 2.1: The definition of natural numbers as an inductive data type.

The definition of naturals here looks a lot like the GADT (generalised algebraic data type [9]) rendition in Haskell would; this is no coincidence. Notice that we have to define that Natural is of type Set. In Agda, Set is the type of types: types are also simply values.

Just as in Haskell, we can also use pattern matching to do operations on natural numbers. Let us look at the definition of addition of natural numbers.

```
_+_ : Natural → Natural → Natural
zero     + m = m
(succ n) + m = succ (n + m)
```

Mixfix Notice how we write _+_ for the name of the function, then later drop the underscores. This notation is referred to as mixfix – in Agda we are allowed to define operators using underscores to denote where they expect arguments. Other than that, addition is fairly straightforward, using the inductive style of programming we will come to know and love.

We will now look at the definition of lists in Agda. This is already starting to look slightly different to the corresponding Haskell implementation.

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

The first thing to note is that we are allowed to use Unicode symbols for function and constructor names – the combination of mixfix and Unicode makes Agda very liberal in what is accepted as an identifier. The next thing to note is that the List data type is parameterised by an argument, A, of type Set. Recall that Set is the type of types, so List is parameterised by a type for the values it should contain.

Our shiny new data types are clamouring to be used, so why not define a head function on List? The function head is supposed to give the first element of a list. A Haskell programmer would probably write something like head_0.

```
head₀ : List A → A
head₀ (x :: xs) = x
```

Telescopes There are a number of problems with the definition of head_0! The first thing Agda will complain about, is that A is undefined in the type signature. The solution is simple: we introduce a *telescope*. A telescope is like a lambda function, except that it is a function on types (also known as values of type Set). The result of this attempt is head_1.

```
head₁ : {A : Set} → List A → A
head₁ (x :: xs) = x
```

Implicit Arguments This is getting somewhere, and we have again introduced a new concept: implicit arguments. The distinction between explicit (usual arguments to functions, as seen in Haskell, for example) and implicit arguments is merely that the latter are tagged as hidden, and do not have to be provided if they can be inferred from the context. Arguments are marked hidden by surrounding them with curly braces in the telescope; in head_1, A is an example of such an argument. This often reduces the number of "obvious" arguments that have to be explicitly passed around, reducing visual clutter. Compare this to the way class constraints on Haskell functions cause a dictionary to be passed around implicitly. In the head_1 example, we need not give the type of the elements of the list, since Agda can infer this information from whatever list we pass.

Totality In spite of our enthusiasm, this definition of head_1 still will not be accepted by Agda. Another important concept is that of *totality*: a function is *total* when it is both terminating and defined on all inputs. All functions are required to be total. Termination is checked by making sure that recursive calls are always done on *structurally smaller* arguments – as is the case in the addition example. Furthermore, a function is considered to be defined on all inputs when the patterns it matches on cover all the possibilities. What we mean by this is that an alternative should be given for each possible constructor – something which is violated by the head_1 attempt: it is missing a case for the empty list. This is something Haskell does not care about; it simply smirks and throws an exception if we try to normalise the expression head [].

If we have to define head_2 for the empty list too, we will have to resort to a Maybe type. The definition of Maybe is omitted, because it is hardly surprising to a Haskell programmer.

```
head₂ : {A : Set} → List A → Maybe A
head₂ []       = nothing
head₂ (x :: xs) = just x
```

Of course, this Maybe is something of an annoyance; it would be preferable to guarantee that the empty list is not valid input to the head_2 function. This is where a dependently typed language really comes into its own. We will now move on to the *de facto* example of a dependent type, the vector – like a list, but with a fixed length.

```
data Vector (A : Set) : Natural → Set where
  []   :                                    Vector A zero
  _::_ : {n : Natural} → A → Vector A n → Vector A (succ n)
```

Indexed data types We now have an argument both to the left and the right of the colon in the type signature of the data definition. Left-hand arguments are called parameters, and scope over all the constructors. Right-hand arguments are called indices, and only scope over single constructors, and as such need to be introduced per constructor using a telescope. The _::_ constructor has a size parameter, which is an example of such an index. This dependent type has the advantage that we can distinguish vectors of different sizes by their type, at compile time, without knowing their value.

Now we can write functions like the following, our final (and most successful) attempt: head_3.

```
head₃ : {A : Set} {n : Natural} → Vector A (succ n) → A
head₃ (x :: xs) = x
```

We see that only vectors with a value n such that succ n is the length of the vector, are valid inputs. This way, we guarantee that empty vectors cannot be beheaded. Agda is also convinced that this function is total, so we are done: we have a safe head function.

This is probably the most common example of why DTP[5] is the best thing since sliced bread: we cannot ask for the head of an empty vector, since we will get a compile time error that there is no possible value of n such that succ n ≡ zero. Compare this to the head function defined in Haskell's Prelude, where a run time exception is generated if an empty list is passed in. How primitive!

Now that we have seen the basics of Agda, we will introduce a few tricks used in the rest of this project which may not be completely intuitive.

2.2 More on Pattern Matching

So far we have seen the very basics of Agda. A few aspects deserve more attention, though, one prime example being Agda's pattern matching facilities.

One of Haskell's selling points is the ability to do pattern matching. This makes writing structurally recursive functions both easy and the Natural Way of Doing Things™. Agda shares this idiomatic programming style, but has a much more powerful version of pattern matching, namely dependent pattern matching.

2.2.1 Absurd Patterns

Something that the Agda pattern matching system has which is completely different from Haskell, is the idea of absurd patterns. The pattern matching in

[5] DTP stands for *dependently typed programming*.

Agda is dependent. This means that based on the (rich) type information available about terms, certain combinations of arguments are automatically regarded as impossible, or *absurd*, to use Agda lingo.

We will see what this means by way of an example. Another interesting data type is that of finite natural numbers. The usual definition of naturals, Natural, has no maximum value, but we can define a data type, called Fin n, that only contains values smaller than n.

```
data Fin : Natural → Set where
  zero : {n : Natural}              → Fin (succ n)
  succ : {n : Natural} (i : Fin n) → Fin (succ n)
```

We might want to convert a value in Fin n to a value in Natural, since every value that can be expressed in Fin n has a cousin in Natural. The function natural performs this conversion. Note the use of the absurd pattern, `()`.

```
natural : (n : Natural) → Fin n → Natural
natural zero     ()
natural (succ n) zero     = zero
natural (succ n) (succ m) = succ (natural n m)
```

Basically, the *dependent* in dependent pattern matching refers to the fact that given the specialisation of the function in the case where zero is the first argument, it can be inferred that the next argument should be of type Fin zero, which obviously has no inhabitants (no natural numbers are strictly smaller than zero). This is why we need not provide this branch.

We now can convince Agda that even though there are no alternatives provided for the n ≡ zero case, the function is still total.

This is something which is not necessary in Haskell, since we are not required to write total functions. There, we are left to our own devices, and should be responsible programmers that do not write code that may trigger pattern matching failures.

2.2.2 Inferable Patterns

Another Agda pattern matching feature we will often encounter is dotted patterns. Because pattern matching is dependent, information about certain arguments can often be inferred from others. Using a dotted pattern means certain parameters are inferable or equal to others.

For purposes of illustration, we will define an equality type.

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

The type Equal only contains values constructed using refl (which stands for reflexive), and refl can only be used when the arguments to Equal are identical. This is because the same x is used as both first and second argument to Equal.

We might use the equality type as follows; we are writing a function which is only defined on equal naturals. Here we pattern match on whether some naturals are equal, and if so, we can use this information on the left-hand side of the equation too. Note how repeated variables on the left-hand side are allowed, if their value is inferable.

```
weird : (n  m : Natural) → Equal n m → List Natural
weird    .m m             refl       = zero :: []
```

Other than the features explicitly mentioned in this chapter so far, the usual constructs such as records, modules and let-bindings are present and behave as expected. Bear in mind that type signatures may also include let-bindings. Definitions of functions and data types are also very similar to those found in Haskell, except that in contrast with Haskell, you have to use GADT-like notation for data constructors.

Agda is, practically speaking, like Haskell with a type system on steroids[6]. The discussion of how and why this is so is considered out of the scope of this project, but suffice it to say that tearing down the distinction between values and types allows powerful new techniques, such as invariant-guaranteeing data types. We will see many examples of these in the following chapters.

After Haskell though, looking at Agda for the first time can be confusing, since a number of foreign concepts are introduced. In the remainder of this chapter, we will pay attention to a number of tricks, the utility or sense of which might not at first be apparent.

2.3 A Programming Language *and* Proof Assistant

It has already been briefly mentioned that Agda is both a proof assistant and a programming language, as a result of the Curry–Howard isomorphism. This correspondence defines a relationship between programs as proofs and types as theorems.

In this section, I will give a short explanation of how this correspondence works, and what it means for programmers. I refrain from attempting to give a comprehensive explanation of intuitionistic logic and why the exact correspondence between natural deduction and simply typed λ-calculus exists. The disillusioned reader is advised to take a look at the Lectures on the Curry–Howard isomorphism by Sørensen and Urzyczyn [53].

Programs as Proofs Intuitionistic logic is at the heart of Agda as a proof assistant. It is similar to classical logic, and works as expected (including implication, conjunction, etc.), but there is a big difference: $A = \neg\neg A$ is not a theorem[7]. In intuitionistic logic, only once one provides a constructive proof of a proposition, is it regarded as a theorem.

By the Curry–Howard isomorphism, propositions are types. A logical proposition, such as $a \Rightarrow b$ translates to a type a → b. Now, $a \Rightarrow b$ is a theorem if

[6] For example, the Agda type system does β-reduction on terms – evaluation – which is something seen as possible, but quite exotic, in Haskell-land.

[7] In mathematical parlance, a *theorem* is a true and proven proposition, whereas a *nontheorem* is a false proposition.

and only if the type `a → b` is inhabited. One of the most intuitive illustrations of this is the proposition $A \Rightarrow B \Rightarrow A$. This can be read as "if we assume A is true, and we furthermore assume that B is true, then we can conclude A." Obviously this is a theorem, but let us translate the proposition to a type. It becomes `a → b → a`; an example of a function with such a type is `const`, the function that returns its first argument regardless of what its second is. Clearly, if we see the first argument with type `a` as a proof of A, then it is obvious that regardless of what is given as the second argument, we fulfil our proof obligation `a` by returning the first argument unchanged. It was, after all, a proof of A.

Keeping this correspondence in mind, we can give analogues of mathematical logic in type theory. The trivial theorem, `true`, translates to the type $\top$, which has one inhabitant, `tt`. The simplest nontheorem `false` translates to $\bot$, the type with no inhabitants. Therefore, a proof for $\bot$ can never be constructed. Other equivalents are $\_\wedge\_$ and $\_\times\_$ (`Pair` in Haskell), which are only proven if both left and right components are inhabited. Disjunction (a.k.a. the $\_\vee\_$ operator in logic) translates to the $\_\uplus\_$ data type (known as `Either` in Haskell), which has constructors for left *or* right. This interpretation of types as propositions is also known as the Brouwer–Heyting–Kolmogorov interpretation, and was proposed by Brouwer and Heyting and independently by Kolmogorov [57]. For an interesting history of logic in computer science, and a clear explanation of the Curry–Howard isomorphism, Wadler's article is a good bet [59].

Now that we have an intuition for the Curry–Howard isomorphism, we can continue looking at various aspects of Agda as a proof assistant. One point worth noting is that in Agda, one directly manipulates and constructs proof objects in the same language as is used to express computation. In many other theorem proving systems, such as Coq, there is a separate tactic language for writing proofs [12]. However, both systems are based on intuitionistic logic, therefore the same concepts hold.

Termination In the previous section, the necessity of defining total functions was mentioned. This is no arbitrary choice, for without this property, Agda's logic would not be sound. Not enforcing the termination aspect of totality would make it easy to define a proof of falsity, as we have done in the function `falsity`.

```
falsity : ⊥
falsity = falsity
```

This is why the termination criterion is that at least one of the arguments to the recursive call be *structurally smaller*. Compare the addition of naturals example where `suc n` is pattern matched, and `n` is passed as a recursive argument – `n` is indeed structurally smaller than `suc n`.

Covering Being defined on all possible inputs is also an aspect of totality. If this requirement were dropped, a number of desirable properties for a logic would not hold any longer. The most obvious example is that all of a sudden, run time exceptions are possible: if a function is not defined on a given input but we apply that argument anyway, bad things will happen (compare Haskell and a run time pattern matching failure). Because functions can also return types (which are also simply values) and thus be used in type signatures, we would

not want it to be possible for type checking to break as a result of an incomplete function definition.

Finally, Agda allows us to define functions and proofs side-by-side, allowing concurrent development of programs and proofs of properties about those programs hand-in-hand. The Emacs mode, which is typically used to interactively develop proofs and programs, has a concept of *holes* – we are free to place a question mark anywhere in the file, and compile. This question mark turns into something which looks a bit like `{ }n`, which we call a goal. When the cursor is placed inside a goal, queries such as the type of the value expected there or the objects in the environment at that point are available.

Admittedly, this section is by no means a comprehensive explanation of the Curry–Howard isomorphism. More information about how to use Agda as a proof assistant is available [43, 12]. For background reading on the Curry–Howard isomorphism a reference should again be made to Sørensen *et al.* [53]. We will now look at some tricks which are peculiar to Agda; hopefully dealing with those now will make code snippets introduced later a little more comprehensible.

2.4 Implicit Record-type Arguments

Previously, in Sec. 2.1, we saw how certain arguments which are considered obvious, can be marked implicit. This technique makes calls to functions more concise, since some arguments are not explicitly listed. When the value of an argument can be inferred, this technique becomes particularly useful. Furthermore, it turns out that records have advantageous properties when it comes to inferring their values. This section demonstrates the technique.

If a particular argument is a record, and it has only one possible inhabitant, Agda can automatically infer its value. Thus, it also need not be passed as an explicit argument at the call-site. The code snippet in Fig. 2.2 illustrates how record-type arguments having only one alternative can be automatically inferred.

```
foo : ⊤ × ⊤ → ℕ
foo u = 5
bar : ℕ
bar = foo _
```

Figure 2.2: Illustrating the automatic inference of record-type arguments. Note that it is possible to replace u in foo with the irrefutable pattern u_1, u_2, since, as has been mentioned before, this is the only valid constructor for the type $\_\times\_$.

The function foo expects a value of type $\top \times \top$, and returns a natural number. We know, however, that $\_\times\_$ is a record and only has the constructor $\_,\_ : A \to B \to A \times B$. Therefore, the only possible value is one using the constructor $\_,\_$. If we next look at the values for A and B here, namely the left and right-

hand arguments' types, we see that in both cases they have type $\top$. The unit type is also defined as a record with only one constructor, namely tt. This means that the only value possible is tt, tt, which is why we can use the underscore notation, meaning Agda should infer the argument for us.

The fact that pairs and unit are defined as records in the standard library is pretty crucial here. The type system does some work for us in these cases; η-reduction is done on record types, which allows Agda to infer that there is exactly one inhabitant of a certain type. This η-reduction is not done on general data types, since this would increase the complexity of the work the compiler needs to do as well as potentially introduce unsound behaviour [11].

Also, it means that it is possible to assert to Agda that values of a certain type are always inhabited. We call this assertion an *irrefutable pattern*, see Fig. 2.3. Here, we pattern match on (tt, tt), and Agda is convinced that no other options are possible.

Since inference is possible, we can also make this argument implicit, which effectively hides from the user that a value is being inferred and passed, as in Fig. 2.3. This saves us an underscore.

```
foo : {u : ⊤ × ⊤} → ℕ
foo {tt , tt} = 5
bar : ℕ
bar = foo
```

Figure 2.3: Implicit (or hidden) arguments are inferred, if possible.

This is possible, since the type $\top \times \top$ only has one inhabitant. If multiple values were valid, the above code would have resulted in an unsolved meta[8] in the definition of bar. That brings us to one of the drawbacks of this solution which has been used quite often. Mainly, the technique has been used to "hide" a proof witness of, for example, an input term being of the right shape. The problem with this trick is that if an implicit argument is ambiguous, or worse, if it is a type with no inhabitants[9], the compiler will not fail with a type error, but merely with a warning that there is an unsolved meta. The corresponding piece of code will be highlighted yellow in the Emacs Agda mode, but the user will not be given any fatal error. The problem is then that an inattentive programmer might miss this innocuous-looking error, while it actually represents an error in a proof. Luckily Agda prevents us from importing modules with unsolved metas, mitigating the danger of hiding proofs this way.

Now that we have seen some idiosyncrasies which could otherwise cause confusion later on, it is time to move on to the real reason for introducing Agda. Let us start using the possibilities we have thanks to dependent types! Of course, a full introduction to the Agda language including all its curiosities and features is out of the scope of such a crash course. In closing, the inquisitive reader is invited to work through Norell's excellent tutorial [43].

[8] An unsolved meta can be thought of as an argument which cannot be inferred.
[9] A type with no inhabitants represents a false proposition.

Chapter 3

Reflection in Agda

Since version 2.2.8, Agda includes a reflection API [1], which allows converting parts of a program's code into abstract syntax, in other words a data structure in Agda itself, that can be inspected or modified like any other data structure. The idea of reflection is old: already in the 1980s Lisp included a similar feature, called *quoting* and *unquoting*, which allowed run time modification of a program's code, by the program itself. This has given rise to powerful techniques for reusing code and generating frequently needed but slightly different expressions automatically. What can be done with Lisp, can be done better using Agda; at least, so we hope. This chapter looks at the current state of the admittedly work-in-progress reflection API, and illustrates how to use it. It should be a good starting point for someone already comfortable with Agda to find inspiration on how to make reflection work to their advantage.

Agda's reflection API defines several data types which represent terms, types, and sorts. These definitions take into account various features, including hidden arguments and computationally irrelevant terms. An overview of the core data types involved is included in Sec. 3.2. In addition to these data types that represent *terms*, there is limited support for reflecting *definitions*. Inspection of definitions is detailed in Sec. 3.2.1. Continue reading Sec. 3.1 for a practical guide to reflection.

3.1 The Basics

Before going into too much detail about how reflection works and what data types are involved, we will look at a few simple code snippets which should serve to illustrate the basics of using the reflection API.

Caveat One rather serious word of admission is to be made here. The code presented in this thesis does not work out of the box as advertised. For this code to compile, some minor changes to the Agda compiler are necessary. For reasons which will be made clear in Chapter 5, the abstract data type representing terms inside the Agda compiler (the one in Fig. 3.1) needed to be extended with an extra argument to the constructor representing a lambda abstraction, denoting the type (or more accurately, a representation thereof in terms of `Type`) of the argument bound in that abstraction. There is a high likelihood that the changes

to the Agda reflection API detailed in Appendix A.1 will be adopted in a future version of Agda, but at the time of writing a personal fork of the compiler's repository was used[1].

The Keywords There are several new keywords that can be used to quote and unquote Term values: quote, quoteTerm, quoteGoal, and unquote. The quote keyword allows the user to access the internal representation of any identifier, or name. This internal representation can be used to query the type or definition of the identifier.

The easiest example of quotation uses the quoteTerm keyword to turn a fragment of concrete syntax into a Term value. Note that the quoteTerm keyword reduces like any other function in Agda. As an example, the following unit test type checks:

```
example₀ : quoteTerm (λ (x : Bool) → x)
         ≡ lam visible (el _ (def (quote Bool) [])) (var 0 [])
example₀ = refl
```

Dissecting this, we introduced a lambda abstraction, so we expect the lam constructor. Its one argument is visible, and the body of the lambda abstraction is just a reference to the nearest-bound variable, thus var 0, applied to an empty list of arguments.

The el constructor The test example₀ also shows us that the quoted lambda binds a variable with some type. The el constructor we see represents the type of the argument to the lambda. The first argument to el represents the sort, if it is known. In example₀, it is unknown. Furthermore the type of x is Boolean, represented as def (quote Bool) []. This means the Bool type (which is a definition, hence def) with no arguments.

Furthermore, quoteTerm type checks and normalises its term before returning the required Term, as the following example demonstrates:

```
example₁ : quoteTerm ((λ x → x) 0) ≡ con (quote ℕ.zero) []
example₁ = refl
```

See how the identity function is applied to zero, resulting in only the value zero. The quoted representation of a natural zero is con (quote zero) [], where con means that we are introducing a constructor. The constructor zero takes no arguments, hence the empty list.

The quoteGoal keyword is slightly different. It is best explained using an example:

```
exampleQuoteGoal : ℕ
exampleQuoteGoal = quoteGoal e in { }₀
```

[1]This fork, along with a version of the Agda standard library with the modifications necessary to work with it, is available at `https://darcs.denknerd.org`.

The quoteGoal keyword binds the variable e to the Term representing the type of the current goal. In this example, the value of e in the hole will be def ℕ [], i.e., the Term representing the type ℕ.

Another function that deals with types is the aptly named type function. Given a Name, such as the result of quote example$_0$, type returns the Type value representing the type of that identifier. This indeed implies one cannot ask the type of an arbitrary Term, since one would need to introduce it as a definition first, to be able to get a Name associated with it. In example$_2$ we see what type returns when asked about the successor function (a function with type ℕ → ℕ), and in example$_3$ we verify that the term shown is in fact the same as a function from naturals to naturals. The el constructor is illustrated clearly here. The first argument to el is the sort of the type, where in example$_2$ the lit 0 term denotes a type in Set$_0$ (which is equal to Set). The second argument to el is, as we already saw, the Term-representation of the type.

```
example₂ : type      (quote ℕ.suc)
         ≡ el (lit 0) (pi (arg visible relevant
                              (el (lit 0) (def (quote ℕ) [])))
                              (el (lit 0) (def (quote ℕ) []))))
example₂ = refl
example₃ : type      (quote ℕ.suc)
         ≡ el (lit 0) (quoteTerm (∀ (n : ℕ) → ℕ))
example₃ = refl
```

The unquote keyword converts a Term data type back to concrete syntax. Just as quoteTerm and quoteGoal, it type checks and normalises the Term before it is spliced into the program text.

This short introduction should already be enough to start developing simple reflective programs. The rest of this chapter goes into more detail regarding the data structures involved in Agda's reflection API, and later, gives a detailed account of real-world use-case.

3.2 The Structures of Reflection

After having seen an informal introduction to practical reflection, we will look at the data types involved in reflection. After all, it is a good idea to be aware of what values one might expect as a result from quoteTerm. The full definitions of Term, Type and their helpers are presented in Fig. 3.1.

The first structure we will look at step-by-step is Term, which represents concrete Agda terms.

A variable has a De Bruijn index, represented by a natural number, and may be applied to arguments. The constructors con and def are introduced for constructors and definitions, respectively, applied to a list of arguments. Lambda abstractions bind one variable. Included is the type signature of the argument, represented by a Type. The pi constructor represents function types, or telescopes (the dependent equivalent of an arrow). It can be seen as a lambda abstraction for types instead of terms. Finally the constructor unknown stands

```
postulate Name : Set

-- Arguments may be implicit, explicit, or inferred
data Visibility     : Set where
  visible hidden instance : Visibility

-- Arguments can be relevant or irrelevant.
data Relevance    : Set where
  relevant irrelevant : Relevance

-- Arguments.
data Arg A        : Set where
  arg : (v : Visibility) (r : Relevance) (x : A) → Arg A
-- Terms.
mutual
  data Term       : Set where
    -- A bound variable applied to a list of arguments
    var       : (x : ℕ) (args : List (Arg Term))       → Term
    -- Constructor applied to a list of arguments
    con       : (c : Name) (args : List (Arg Term))    → Term
    -- Identifier applied to a list of arguments
    def       : (f : Name) (args : List (Arg Term))    → Term
    -- Lambda abstraction (typed – see Appendix A.1).
    lam       : (v : Visibility) (σ : Type) (t : Term) → Term
    -- Dependent function types
    pi        : (t₁ : Arg Type) (t₂ : Type)            → Term
    -- Sorts
    sort      : Sort                                   → Term
    -- Anything else
    unknown   :                                          Term
  data Type       : Set where
    el        : (s : Sort) (t : Term)                  → Type
  data Sort       : Set where
    -- A Set of a given (possibly neutral) level.
    set       : (t : Term) → Sort
    -- A Set of a given concrete level.
    lit       : (n : ℕ)    → Sort
    -- Anything else.
    unknown   :              Sort
```

Figure 3.1: The data types for reflecting terms.

for things which are not or cannot be represented in this AST[2], such as function definitions or holes.

As explained in the previous section, the el constructor constructs values in Type. It has two arguments: one for the sort of the type, the other for the Term representing the type.

Aside from the necessary data structures, the Reflection module of the Agda standard library[3] also exports a number of functions. We provide a list of them in Fig. 3.2, along with a description of their use.

```
_≟-Name_ : Decidable {A = Name} _≡_
  -- The other decidable properties are omitted for
  -- brevity, but are similarly named.
type        : Name      → Type
definition  : Name      → Definition
constructors : Data-type → List Name
```

Figure 3.2: The functions exported by the Reflection module of the Agda standard library, as of version 0.6.

The definition function returns the definition of a given identifier. The type Definition is defined as follows.

```
data Definition : Set where
  function    : Function  → Definition
  data-type   : Data-type → Definition
  record′     : Record    → Definition
  constructor′ :            Definition
  axiom       :             Definition
  primitive′  :             Definition
```

At the time of writing the only constructor we can do anything with is data-type: using it we can get a list of constructors, by calling the suitably named constructors function. See the illustration in Sec. 3.2.1.

Finally, we have decidable equality on the following types: Visibility, Relevance, List Args, Arg Types, Arg Terms, Names, Terms, Sorts and Types. Typically, this is useful for deciding which constructor is present in some expression, by comparing to known Names. Such a comparison is illustrated in the function convert, below.

```
convert : Term → Something
convert (def c args) with c ≟-Name quote foo
... | yes p = { }0  -- foo applied to arguments
... | no ¬p = { }1  -- a function other than foo
```

[2]AST stands for *abstract syntax tree*; this abbreviation will be used hereafter.

[3]The standard library version 0.6 was used here; later versions might expose more functionality.

Aside from these functions and types, the Reflection module also contains a few lemmas for decidable equality on terms and types. These are rather boring, and the user will probably never have to use them directly.

3.2.1 Inspecting Definitions

With the functions provided by Reflection we can get a little more insight into definitions of data types. For example, we can get a list of constructors for some data type. The following code snippets illustrate how this is done, and what the format of the answer is.

```
giveDatatype : (d : Definition) → {pf : isDatatype d} → Data-type
giveDatatype (data-type d)      {_}    = d
giveDatatype (function  x)      {()}
...
```

The helper function giveDatatype assumes that the constructor present is, in fact, data-type, which saves some elimination of uninteresting cases. With this helper, we can get the Data-type to feed to the constructors function. The following unit test shows an example, where we ask for all the constructors of the natural numbers.

```
ℕcons : List Name
ℕcons = constructors (giveDatatype (definition (quote ℕ)))
consExample : ℕcons ≡ quote ℕ.zero ::
                      quote ℕ.suc  :: []
consExample = refl
```

Now we have in ℕcons a list of the names of the constructors of the data type ℕ, which we could use to do more interesting things depending on the structure of a data type. One example might be to compute a generic representation which is isomorphic to the naturals, as is often done using Template Haskell. For example, in the Regular library for generic programming [58], a translation to a sum-of-products view is made. This possibility is explored in Chapter 6.

That wraps up all the functionality available from the reflection API. Contemplating what we might want to do using these new tools, it becomes clear that a common task will be casting a raw Term into some AST of our own. I developed a library, Autoquote, which might serve as both an instructive example in how to pull apart Terms, as well as a useful and reusable function, since it can automatically convert a Term into some AST type. All that is needed is a mapping from concrete Agda Names to constructors of this AST. An explanation of its implementation application is given in Sec. 3.3, and an example use-case is given in 4.3.1.

3.3 Automatic Quoting

If, each time we wanted to quote a term, we had to write a huge function, with many pattern matching cases and nested **with** statements to handle different

shapes of ASTs, we would quickly become discouraged. This nearly happened while doing this project, which is why Autoquote was conceived. Quoting some expression with a given grammar is a mundane task we are frequently faced with if we are foolhardy enough to use reflection. The (partial) solution to this problem – something which at least mitigates the agony – is presented in this section.

Imagine we have some AST, for example Expr, in Fig. 3.3. This is a rather simple inductive data structure representing terms which can contain Peano style natural numbers, variables (indexed by an Agda natural) and additions.

```
data Expr : Set where
  Var : ℕ            → Expr
  Pl  : Expr → Expr → Expr
  S   : Expr         → Expr
  Z   :                Expr
```

Figure 3.3: The toy expression language Expr. We would like support for automatically quoting such terms.

We might conceivably want to convert a piece of Agda concrete syntax, such as $5 + x$, to this AST, using reflection. This typically involves ugly and verbose functions like the one from Sec. 4.2 with many **with** clauses and frankly, too much tedium to be anything to be proud of.

We need to check that the Term has a reasonable shape, and contains valid operators. Ideally, we would provide a mapping from concrete constructs such as the _+_ function to elements of our AST, and get a conversion function for free. This motivated my development of Autoquote in the course of this project. What Autoquote does is abstract over this process, and provide an interface which, when provided with such a mapping, automatically quotes expressions that fit. Here, *fitting* is defined as only having variables, or names that are listed in this mapping. Other terms are rejected. The user provides an elegant-looking mapping and Autoquote automatically converts concrete Agda to simple inductive types. The mapping table for Expr is shown in Fig. 3.4.

```
exprTable : Table Expr
exprTable = (Var,
  2 # (quote _+_)   ↦ Pl ::
  0 # (quote ℕ.zero) ↦ Z  ::
  1 # (quote ℕ.suc)  ↦ S  :: [])
```

Figure 3.4: The mapping table for converting to the imaginary Expr AST.

How this should be interpreted is that any variables encountered should be stored as Vars, and the _+_ operator should be a Pl constructor. In each case we are required to manually specify the arity of the constructor: how many

arguments it expects. A zero, from the Data.Nat standard library, should be treated as our Z constructor, and a suc translates to S. These constructors expect 0 and 1 argument, respectively.

We will now look at the implementation of this library.

Implementation The type Table a, in Fig. 3.5, is what we use for specifying what the AST we are expecting should look like. The function N-ary provides a way of storing a function with a variable number of arguments in our map, and _$ⁿ_ is how we apply the "stored" function to a Vec n of arguments, where n is the arity of the function. Note that this is a copy of the standard library Data.Vec.N-ary, but has been instantiated here specifically to contain functions with types in Set. This was necessary, since the standard library version of N-ary can hold functions of arbitrary level (i.e. Set n). Therefore, the level of the N-ary argument inside ConstructorMapping could not be inferred (since this depends on which function one tries to store in that field). This yields an unsolved constraint which prevented the module from being imported without using the unsound type-in-type option.

Using this N-ary we can now define an entry in our Table as having an arity, and mapping a Name (which is Agda's internal representation of an identifier, see Fig. 3.1) to a constructor in the AST to which we would like to cast the Term. The definition of N-ary restricts the possible function types to zero or more arguments of type A to an element of type B. In ConstructorMapping, we further specialise this function to zero or more arguments of type astType to astType, which forces us to stick to simple inductive types, such as our Expr example.

```
N-ary : (n : ℕ) → Set → Set → Set
N-ary zero    A B = B
N-ary (suc n) A B = A → N-ary n A B

_$ⁿ_ : ∀ {n} {A : Set} {B : Set} → N-ary n A B → (Vec A n → B)
f $ⁿ []       = f
f $ⁿ (x :: xs) = f x $ⁿ xs

data ConstructorMapping (astType : Set) : Set₁ where
  _#_↦_ : (arity : ℕ)
        → Name
        → N-ary arity astType astType
        → ConstructorMapping astType

Table : Set → Set₁
Table a = (ℕ → a) × List (ConstructorMapping a)
```

Figure 3.5: The types and helper functions associated with the Autoquote library.

With the above ingredients we can now define the function convert shown in Fig. 3.6. It takes a mapping of type Table a, and a Term obtained from one of Agda's reflection keywords, and produces a value which might be a prop-

erly converted term of type a. Here, a is the type we would like to cast to, for example Expr. We also have the helper function lookupName, which finds the corresponding entry in the mapping table. If nothing usable is found, nothing is returned.

An example of such a mapping would be the one required for our Expr example, presented in Fig. 3.4.

Note that convert is not intended to be called directly; a convenience function doConvert is defined later.

```
lookupName : {a : Set} → List     (ConstructorMapping a)
                       → Name
                       → Maybe (ConstructorMapping a)
mutual
  convert : {a : Set} → Table a → Term → Maybe a
  convert (vc, tab) (var x args)  = just (vc x)
  convert (vc, tab) (con c args)  = appCons (vc, tab) c args
  convert (vc, tab) (def f args)  = appCons (vc, tab) f args
  convert (vc, tab) _             = nothing
```

Figure 3.6: The function convert.

If convert encounters a variable, it just uses the constructor which stands for variables. Note that the parameter is the De Bruijn index of the variable, which might or might not be in scope. This is something to check for afterwards, if a just value is returned.

In the case of a constructor or a definition applied to arguments, the function appCons is called, which looks up a Name in the mapping and tries to recursively convert its arguments, then applies the corresponding constructor to these new arguments. Before this is done, the number of arguments is also compared to the defined arity of the function.

The function convertArgs takes a list of term arguments (the type Arg Term) and tries to convert them into a list of AST values.

```
appCons : {a : Set} → Table a → Name → List (Arg Term) → Maybe a
appCons (vc, tab) name args with lookupName tab name
... | just (arity # x ↦ x₁)          with convertArgs (vc, tab) args
... | just (arity # x₁ ↦ x₂)         | just x   with length x ≟-ℕ arity
... | just (.(length x) # x₁ ↦ x₂)   | just x   | yes
                                                = just (x₂ $ⁿ fromList x)
... | just (arity # x₁ ↦ x₂)         | just x   | no = nothing
... | just (arity # x ↦ x₁)          | nothing = nothing
... | nothing                        = nothing

convertArgs : {a : Set} → Table a → List (Arg Term) → Maybe (List a)
convertArgs tab []                  = just []
convertArgs tab (arg v r x :: ls) with convert tab x
... | just x₁  with convertArgs tab ls
... | just x₂  | just x₁  = just (x₂ :: x₁)
... | just x₁  | nothing = nothing
... | nothing = nothing
```

Note that we will probably need to post-process the output of convert, but this will be illustrated later, in Sec. 4.3.1.

If all of these steps are successful, the converted Term is returned as just e, where e is the new, converted member of the AST. For example, see the unit test in Fig. 3.7. Convenience functions for dealing with failing conversions are also provided. The doConvert function makes the assumption that the conversion succeeds, which enables it to return a value without the just. Furthermore, this assumption, defined in convertManages, is an inferable proof. This is on account of it being a record type, which is explained in Sec. 2.4.

```
convertManages : {a : Set} → Table a → Term → Set
convertManages t term with convert t term
convertManages t term | just x  = ⊤
convertManages t term | nothing = ⊥

doConvert : {a : Set} → (tab : Table a)
                      → (t : Term)
                      → {man : convertManages tab t}
                      → a
doConvert tab t {man} with convert tab t
doConvert tab t {man} | just x = x
doConvert tab t {()   } | nothing
```

The use of convertManages and doConvert is illustrated in Fig. 3.7. This approach, using convertManages as an assumption, is a lot simpler than writing by hand a predicate function with the same pattern matching structure as convert. Adding to the complication, **with** clauses are often expanded unpredictably in practice. The net effect of writing a pair of functions in this style is the same as the "usual" way of writing a predicate function by hand, in that a compile time error is generated if the function doConvert is invoked on an argument with the wrong shape. Compare these relatively elegant functions to the verbose

term2boolexpr and isBoolExprQ functions in Sec. 4.3.

```
something : {x y : ℕ} → doConvert exprTable
                                  (quoteTerm ((1 + x + 2) + y))
                        ≡ S (PI (PI (Var 1)
                                    (S (S Z)))
                                (Var 0))
something = refl
```

Figure 3.7: An example of Autoquote in use. See Fig. 3.4 for the definition of exprTable, a typical Name-to-constructor mapping.

The format of the translation Table required could most probably be made a little simpler, by not requiring the user to provide the arity of the function, but using the tools explained in Sec. 3.2.1 (the section on inspecting data definitions, and specifically the function constructors in combination with type) to try and discover the arity of the various constructors. Because of time constraints, however, this is left as a suggestion for future work on the Autoquote library.

The BoolExpr AST used in Sec. 4.2 provides a good motivating example for using Autoquote, therefore a more realistic example of Autoquote in use can be found in Sec. 4.3.1. One might also use the ability of quoting arithmetic equations shown here in combination with a monoid solver, such as the example in Norell *et al.* [7].

Further examples of Autoquote functionality can be found in the module Metaprogramming.ExampleAutoquote. The module Metaprogramming.Autoquote contains what could serve as a basis for a system for quoting concrete Agda into a more complex user-defined AST. Now that we have had a quick introduction to Agda in Chapter 2, and defined this library, it is time to move on to putting it all to use.

Chapter 4

Proof by Reflection

The idea behind proof by reflection is simple: given that type theory is both a programming language and a proof system, it is possible to define functions that compute proofs. Reflection is an overloaded word in this context, since in programming language technology reflection is the capability of converting some piece of concrete code into an abstract syntax tree object that can be manipulated in the same system. Reflection in the proof technical sense is the method of mechanically constructing a proof of a theorem by inspecting its shape. Here we will see two case studies illustrating proof by reflection and how Agda's reflection mechanism can make the technique more accessible.

4.1 Simple Example: Evenness

Sometimes, the best way to explain a complicated topic is to start by giving some simple examples. Proof by reflection is no different: it is not a difficult technique, but can initially be counter intuitive.

To illustrate the concept of proof by reflection, we will cover an example taken from Chlipala [10], where a procedure is developed to automatically prove that a number is even. We start by defining the property Even below. There are two constructors: the first constructor says that zero is even; the second constructor states that if n is even, then so is $2 + n$.

```
data Even  : ℕ → Set where
  isEven0  :                    Even 0
  isEven+2 : {n : ℕ} → Even n → Even (2 + n)
```

Using these rules to produce the proof that some large number n is even is tedious: the proof that $2 \times n$ is even requires n applications of the isEven+2 constructor. For example, here is the proof that 6 is even:

```
isEven6 : Even 6
isEven6 = isEven+2 (isEven+2 (isEven+2 isEven0))
```

To automate such proofs, we will show how to *compute* the proof required.

We start by defining a predicate even? that returns the unit type when its input is even and bottom otherwise. In this context, $\top$ and $\bot$ can be seen as the analogues of true and false, just as presented in Sec. 2.3. The meaning of such a decision function is that there exists a proof that some number is even, if it is 0 or $2 + n$, for even n. Otherwise, no proof exists. We will have to prove this, though. The idea of "there exists" is perfectly modelled by the unit and empty types, since the unit type has one inhabitant, the empty type none.

```
even? : ℕ → Set
even? 0             = ⊤
even? 1             = ⊥
even? (suc (suc n)) = even? n
```

Next we need to show that the even? function is *sound*; that our claim holds. To do so, we prove that when even? n returns $\top$, the type Even n is inhabited, and since we are working in a constructive logic, the only way to show this is to give some witness. This is done in the function soundnessEven. What we are actually doing here is giving a recipe for constructing proof trees, such as the one we manually defined for isEven6.

```
soundnessEven : {n : ℕ} → even? n → Even n
soundnessEven {0}           tt = isEven0
soundnessEven {1}           ()
soundnessEven {suc (suc n)} s  = isEven+2 (soundnessEven s)
```

Note that in the case branch for 1, we do not need to provide a right-hand side of the function definition. The assumption, even? 1, is uninhabited, and we discharge this branch using Agda's absurd pattern, `()`.

Now that this has been done, if we need a proof that some arbitrary n is even, we only need to call soundnessEven. Note that the value of n is an implicit argument to soundnessEven. The only argument we need to provide to our soundnessEven lemma is a proof that even? n is inhabited. For any closed term, such as the numbers 28 or 8772, this proof obligation reduces to $\top$, which is proven by the single constructor it has, tt.

```
isEven28   : Even 28
isEven28   = soundnessEven tt
isEven8772 : Even 8772
isEven8772 = soundnessEven tt
```

Now we can easily get a proof that arbitrarily large numbers are even, without having to explicitly write down a large proof tree. Note that it is not possible to write something with type Even 27, or any other uneven number, since the parameter even? n is equal to $\bot$, thus tt would not be accepted where it is in the Even 28 example. This will produce a $\top$!=< $\bot$ type error at compile time. Note that it is possible to generate a user-friendly "error" of sorts, by replacing the $\bot$ constructor in even? with a type with a descriptive name such as `NotEven`.

Of course it should still be an empty type, but possibly parameterised with a natural to indicate which value is odd. This makes the soundness proof a little less straightforward, but in return the type error generated if an odd number is used becomes more informative. This enhancement is demonstrated in Fig. 4.2, in the Boolean tautologies example.

Since the type ⊤ is a simple record type, Agda can infer the tt argument, as explained in Sec. 2.4. This means we can turn the assumption even? n into an implicit argument, so a user could get away with writing just soundnessEven as the proof, letting the inferrer do the rest. For the sake of exposition this is not done here, but the final implementation available on GitHub does make use of this method. A detailed explanation of this technique, which is used extensively in the final code, is given in Sec. 2.4. Note that it still has the minor danger of making errors look like innocuous warnings.

An implementation of the above, including detailed comments, is to be found in the module Proofs.IsEven.

This concludes the example of proving that certain naturals are even using proof by reflection. The next step will be to use the same approach for a more involved and realistic problem.

4.2 Second Example: Boolean Tautologies

Obviously, the first example of proof by reflection, the evenness of natural numbers, was a rather trivial one. There was a good reason for studying it, though, since we will now apply the same technique to a more interesting problem, making the relationship to the previous example clear at each step.

Another application of proof by reflection is Boolean expressions which are a tautology. We will prove this by evaluation of the formulae. We will follow the same recipe as for even naturals, with one further addition. In the previous example, the input of our decision procedure even? and the problem domain were both natural numbers. As we shall see, this need not always be the case: more complex structures and properties may be used.

Take as an example the Boolean formula in equation 4.1.

$$(p_1 \vee q_1) \wedge (p_2 \vee q_2) \Rightarrow (q_1 \vee p_1) \wedge (q_2 \vee p_2) \tag{4.1}$$

It is trivial to see that this is a tautology, but proving this using deduction rules for classical logic would be rather tedious. It is even worse if we want to check if the formula always holds by trying all possible variable assignments, since this will give 2^n cases, where n is the number of variables.

To automate this process, we will follow a similar approach to the one given in the section on even natural numbers (Sec. 4.1). We start by defining an inductive data type to represent Boolean expressions with at most n free variables in Fig. 4.1.

There is nothing surprising about this definition; we use the type Fin n to ensure that variables (represented by Atomic) are always in scope. If we want to evaluate the expression, however, we will need some way to map variables to values. Enter Env n: it has fixed size n since a BoolExpr n has at most n free variables.

```
data BoolExpr (n : ℕ) : Set where
  Truth      :                                  BoolExpr n
  Falsehood  :                                  BoolExpr n
  And        : BoolExpr n → BoolExpr n →        BoolExpr n
  Or         : BoolExpr n → BoolExpr n →        BoolExpr n
  Not        : BoolExpr n →                     BoolExpr n
  Imp        : BoolExpr n → BoolExpr n →        BoolExpr n
  Atomic     : Fin n →                          BoolExpr n
```

Figure 4.1: Inductive definition of Boolean expressions with n free variables.

```
Env : ℕ → Set
Env = Vec Bool
```

Now we can define a decision function, which tells us if a given Boolean expression is true or not, under some assignment of variables. It does this by evaluating the formula's AST, filling in for Atomic values the concrete values which are looked up in the environment. For example, And is converted to the Boolean function _∧_, and its two arguments in turn are recursively interpreted.

```
⟦_⊢_⟧ : ∀ {n : ℕ} (e : Env n) → BoolExpr n → Bool
⟦ env ⊢ Truth       ⟧ = true
⟦ env ⊢ Falsehood   ⟧ = false
⟦ env ⊢ And be be₁  ⟧ =   ⟦ env ⊢ be ⟧ ∧ ⟦ env ⊢ be₁ ⟧
⟦ env ⊢ Or  be be₁  ⟧ =   ⟦ env ⊢ be ⟧ ∨ ⟦ env ⊢ be₁ ⟧
⟦ env ⊢ Not be      ⟧ = ¬ ⟦ env ⊢ be ⟧
⟦ env ⊢ Imp be be₁  ⟧ =   ⟦ env ⊢ be ⟧ ⇒ ⟦ env ⊢ be₁ ⟧
⟦ env ⊢ Atomic n    ⟧ = lookup n env
```

Recall our decision function even? in the previous section. It returned $\top$ if the proposition was valid, $\bot$ otherwise. Looking at $[\![\_\vdash\_]\!]$, we see that we should just translate true to the unit type and false to the empty type, to get the analogue of the even? function.

We call this function So, the string parameter serving to give a clearer type error to the user, if possible.

```
data Error (e : String) : Set where
So : String → Bool → Set
So _   true  = ⊤
So err false = Error err
P  : Bool → Set
P  = So "Argument expression does not evaluate to true."
```

Figure 4.2: Helper type Error, enabling clearer type errors.

Now that we have these helper functions, it is easy to define what it means to be a tautology. We quantify over a few Boolean variables, and wrap the formula in our P decision function. If the resulting type is inhabited, the argument to P is a tautology, i.e., for each assignment of the free variables the entire equation still evaluates to true. An example encoding of such a theorem is Fig. 4.3 – notice how similar it looks to the version expressed in mathematical notation, in equation 4.1.

One might wonder why propositions are not encoded in the slightly more intuitive propositional equality style, for example $(b : \mathsf{Bool}) \to b \vee \neg b \equiv \mathsf{true}$, since that notation more obviously reflects the meaning of "being a tautology", as opposed to one having to understand the So function; this is justified in Sec. 4.2.1.

```
exampletheorem : Set
exampletheorem = (p₁ q₁ p₂ q₂ : Bool) →
  P ((p₁ ∨ q₁) ∧ (p₂ ∨ q₂) ⇒ (q₁ ∨ p₁) ∧ (q₂ ∨ p₂))
```

Figure 4.3: Encoding of an example tautology.

Here a complication arises, though. We are quantifying over a list of Boolean values *outside* of the decision function P, so proving P to be sound will not suffice. We just defined an evaluation function $[\![\_ \vdash \_]\!]$ to take an environment, an expression, and return a Boolean. In Fig. 4.3, though, we effectively quantified over all possible environments. We are going to need a way to lift our decision function to arbitrary environments.

The way we do this is the function foralls, in Fig. 4.4. This function represents the real analogue of even? in this situation: it returns a type which is only inhabited if the argument Boolean expression is true under all variable assignments. This is done by generating a full binary tree of $\top$ or $\bot$ types, depending on the result of $[\![\_ \vdash \_]\!]$ under each assignment. This corresponds precisely to the expression being a tautology if and only if the tree is inhabited.

The Diff argument is unfortunately needed to prove that forallsAcc will eventually produce a tree with depth equal to the number of free variables in an expression.

```
forallsAcc : {n m : ℕ} → BoolExpr m → Env n → Diff n m → Set
forallsAcc b acc (Base   ) = P ⟦ acc ⊢ b ⟧
forallsAcc b acc (Step y) =
  forallsAcc b (true :: acc) y × forallsAcc b (false :: acc) y
foralls : {n : ℕ} → BoolExpr n → Set
foralls {n} b = forallsAcc b [] (zeroleast 0 n)
```

Figure 4.4: The function foralls, which decides if a proposition is a tautology. Compare to the even? function in Sec. 4.1.

What Is This Diff You Speak Of? In Fig. 4.4, we just saw that the Diff argument is necessary. Here, a short description of what it is and why it is needed is given.

The function forallsAcc (among others) has a parameter of type Diff n m. Recalling the function's definition from Fig. 4.4, note that there are two variables, n and m, giving the current size of the environment and the maximum number of bound variables in the proposition, respectively.

This is wrong, since our interpretation function $[\![\_ \vdash \_]\!]$ requires that these m and n are equal. We cannot, however, make them equal in the type signature for forallsAcc, since we are recursively building up the environment with an accumulating parameter. Because of this, we introduce Diff – see Fig. 4.5.

```
data Diff : ℕ → ℕ → Set where
  Base : ∀ {n}                   → Diff n n
  Step : ∀ {n m} → Diff (suc n) m → Diff n m
zeroleast : (k n : ℕ) → Diff k (k + n)
```

Figure 4.5: The definition of the Diff data type.

The Diff data type is necessary because given a term of type BoolExpr m, being a proposition with at most m variables, it should be ensured that in the end an environment of size m is produced. The necessity of $m \equiv n$ is obvious considering that the evaluation function needs to be able to look up the variables in the Boolean expression. As forallsAcc is a recursive function that introduces new variables to the environment one at a time, we need some way to "promise" that in the end m will be equal to n. As can be seen in the definition of the Base constructor, this is exactly what happens.

The same thing is necessary in some of the other functions, given that they also recursively construct or look up proofs that need to have the same size as their BoolExpr argument. Because they use the same technique in a slightly less overt manner they are not separately detailed here.

We also provide the simple lemma zeroleast, which shows that for any two k and n, n steps are needed to count from k to $k + n$.

Soundness Since Diff has been explained, and we now know our real decision function foralls, we can set about proving its soundness. Following the evens example, we want a function something like this.

```
sound : {n : ℕ} → (b : BoolExpr n) → foralls b → ...
```

What should the return type of the sound lemma be? We would like to prove that the argument b is a tautology, and hence, the sound function should return something of the form (b1 ... bn : Bool) → P B, where B is an expression in the image of the interpretation ⟦_⊢_⟧. For instance, the statement exampletheorem is a proposition of this form.

The function proofGoal, given a BoolExpr n, generates the corresponding proof obligation. That is, it gives back the type equivalent to the theorem under scrutiny. It does this by first introducing m universally quantified Boolean variables. These variables are accumulated in an environment. Finally, when m binders have been introduced, the BoolExpr is evaluated under this environment.

```
proofGoal  : (n m : ℕ) → Diff n m → BoolExpr m → Env n → Set
proofGoal .m m (Base   ) b acc = P ⟦ acc ⊢ b ⟧
proofGoal  n  m (Step y) b acc =
           (a : Bool) →
             proofGoal (1 + n) m y b (a :: acc)
```

Now that we can interpret a BoolExpr n as a theorem using proofGoal, and we have a way to decide if something is true for a given environment, we still need to show the soundness of our decision function foralls. That is, we need to be able to show that a formula is true if it holds for every possible assignment of its variables to true or false.

```
soundnessAcc : {m : ℕ} →       (b : BoolExpr m) →
               {n : ℕ} →       (env : Env n) →
               (d : Diff n m) → forallsAcc b env d →
               proofGoal n m d b env
soundnessAcc   bexp env Base H with ⟦ env ⊢ bexp ⟧
soundnessAcc   bexp env Base H | true  = H
soundnessAcc   bexp env Base H | false = Error-elim H
soundnessAcc {m} bexp {n} env (Step y) H =
  λ a → if {λ b → proofGoal (1 + n) m y bexp (b :: env)} a
    (soundnessAcc bexp (true :: env)  y (proj₁ H))
    (soundnessAcc bexp (false :: env) y (proj₂ H))
```

If we look closely at the definition of soundnessAcc, we see that it builds up the environment by assigning some configuration of true and false to the variables. It eventually returns the leaf from foralls which is the proof that the formula is a tautology in that specific case.

```
soundness : {n : ℕ} → (b : BoolExpr n) → foralls b
          → proofGoal 0 n (zeroleast 0 n) b []
soundness {n} b i = soundnessAcc b [] (zeroleast 0 n) i
```

The function soundness calls soundnessAcc with some initial input, namely the BoolExpr n, an empty environment, and the Diff proof that soundnessAcc will be called $(n - 0)$ times. This results in an environment of size n everywhere the expression is to be evaluated.

Now, we can prove theorems by calling soundness b p, where b is the representation of the formula under consideration, and p is the evidence that all branches of the proof tree are true. Agda is convinced that the representation does in fact correspond to the concrete formula, and also that soundness gives a valid proof. In fact, we need not even give p explicitly; since the only valid values of p are nested pairs of tt, the argument can be inferred automatically, if its type is inhabited.

If the module passes the type checker, we know our formula is both a tautology and that we have the corresponding proof object at our disposal afterwards, as in the example of Fig. 4.6.

```
rep        : BoolExpr 2
rep        = Imp (And (Atomic (suc zero)) (Atomic zero))
                 (Atomic zero)
someTauto : (p q : Bool) → P (p ∧ q ⇒ q)
someTauto = soundness rep _
```

Figure 4.6: An example Boolean formula, along with the transliteration to a proposition and the corresponding proof.

Having said that, the trick of letting Agda infer the proof argument to pass to soundness is still a little dangerous, as explained in Sec. 2.4. The thing is, we do not want a user to get away with being able to prove that something which is not a tautology, is a tautology. Since the proof that under all environments the theorem evaluates to true is an inferred argument in this style, one is merely left with an unsolved meta (with an uninhabitable type, to be fair), which might seem a triviality if you do not read the compiler's output carefully. Luckily Agda disallows importing modules with unsolved metas, which means such a spurious proof will not be usable elsewhere in a real-life development.

Other than that potential pitfall, the only part we still have to do manually is to convert the concrete Agda representation (p ∧ q ⇒ q, in this case) into our abstract syntax (rep here). This is unfortunate, as we end up typing out the formula twice. We also have to count the number of variables ourselves and convert them the to De Bruijn indices. This is error-prone given how cluttered the abstract representation can get for formulae containing many variables.

It would be desirable for this process to be automated. In Sec. 4.3 a solution is presented that uses Agda's recent reflection API.

4.2.1 Why Not Propositional Equality?

The question of why the So operator is used here to denote that a formula is a tautology, as opposed to just writing the literal definition of tautology, namely $\forall\ (\mathsf{b} : \mathsf{Bool}) \to \mathsf{Q}\ (\mathsf{b}) \equiv \mathsf{true}$, was asked in the previous section. The reason for this is mainly a technical one. While it is possible to prove tautologies of this form, using this format for reasoning about Boolean formulae becomes rather awkward.

The reason for this is that the So operator returns a type, namely either $\top$, $\bot$ or other record types, which can be passed around as an automatically inferred implicit value (see Sec. 2.4 for a detailed explanation about implicit inferred arguments), removing the need to put refl everywhere such a proof is needed – a unit or pair type can be inferred if it exists[1]. Because of this, the recursive cases of soundness become a lot simpler: the interpretation of a sub-expression being true becomes the same as a unit type being inhabited, and the and-operator corresponds to a pair. If the propositional equality way was being used, many lemmas such as that $\mathsf{a} \wedge \mathsf{b} \equiv \mathsf{true} \Rightarrow \mathsf{a} \equiv \mathsf{true} \wedge \mathsf{b} \equiv \mathsf{true}$ need to be proven, and they are continually needed to pull apart such propositions for recursive calls. Using a type that allows pattern matching with irrefutable patterns to obtain left-truth and right-truth, to then be passed to the recursive calls, is much simpler in this case.

4.2.2 Why Not Enumerate Environments?

A reasonable question to pose, after seeing the interface to the tautology prover, is why we have to separately introduce fresh variables. Why can we not just write something like $\forall\ (\mathsf{e} : \mathsf{Env}\ \mathsf{n}) \to \mathsf{P}$ someprop?

One of the reasons for not enumerating environments is that referring to variables inside someprop becomes a bit of a problem. Some new syntax would have to be introduced, such as a constructor $\mathsf{Var} : \mathsf{Fin}\ \mathsf{n} \to \mathsf{Bool}$ which could be used to refer to an element of the environment by number. This is rather less elegant than the current implementation, which simply brings a few Boolean variables into scope in the native Agda manner, using a telescope (i.e. $(\mathsf{p}\ \mathsf{q}\ \mathsf{r} : \mathsf{Bool}) \to \mathsf{P}\ (\mathsf{p} \wedge \mathsf{q} \Rightarrow \mathsf{r})$, as defined in Sec. 2.1). This has another advantage, namely that when writing down a proposition, you are forced to use only valid variables, which translate to in-scope De Bruijn indices.

Another difficulty of enumerating environments is the generation of the proof goal. Currently, a telescope with Boolean variables can be generated easily via recursion (see the function proofGoal), as opposed to having to generate all possible lists of assignments. Some investigation was done to try and show that environments (lists of Booleans) of length n are enumerable, but the results were not as elegant as those presented here. Also, generating the environments by quantifying over fresh variables and adding them to an accumulating environment saves the hassle of creating a large binary tree with all the possible environments in the leaves.

[1] Compare the example implementation of a ring solver in Agda, which has refls all over the place [31], which cannot be made implicit and thus omitted.

4.3 Adding Reflection

It might come as a surprise that in a project focusing on reflection in Agda, in the programming language technology sense, has not yet found an application for reflection in this chapter. This is about to change. We can get rid of the duplication seen in Fig. 4.6 using Agda's reflection API. In that figure we see the same Boolean formula twice: once in the type signature as an Agda proposition and once in the BoolExpr AST. More specifically, we will use the quoteGoal keyword to inspect the current goal. Given the Term representation of the goal, we can convert it to its corresponding BoolExpr automatically.

The conversion between a Term and BoolExpr is achieved using the function concrete2abstract.

```
concrete2abstract : (t    : Term) → (n : ℕ)
                  → {pf   : isSoExprQ (stripPi t)}
                  → {pf2  : isBoolExprQ n (stripPi t) pf}
                  → BoolExpr n
```

Note that not every Term can be converted to a BoolExpr. Looking at the type signature of the concrete2abstract function, we see that it requires additional assumptions about the Term: it may only contain functions such as _∧_ or _∨_, and bound variables. This is ensured by the predicates isBoolExprQ and friends. The functions stripPi and stripSo remove the quantified variables and P wrapper, respectively.

The concrete2abstract function is rather verbose, and is mostly omitted. A representative snippet is given in Fig. 4.7. The attentive reader will notice that the function in the referenced figure is called term2boolexpr; this is because we also unwrap the outermost call to P and the telescope quantifying over the variables before doing the conversion, since these elements are unnecessary in the BoolExpr representation. The function term2boolexpr can be seen as a helper function to concrete2abstract where the "interesting" work happens. The functions in the type signature, isBoolExprQ and isSoExprQ, simply traverse the Term to see if it fulfils the requirements of being a Boolean expression enclosed in a call to P, preceded by a series of universally quantified Boolean variables.

```
term2boolexpr n (con tf []) pf with tf ≟-Name quote true
term2boolexpr n (con tf []) pf | yes p = Truth
...
term2boolexpr n (def f []) ()
term2boolexpr n (def f (arg v r x :: [])) pf with f ≟-Name quote ¬_
... | yes p = Not (term2boolexpr n x pf)
... | no ¬p with f ≟-Name quote _∧_
...
```

Figure 4.7: The gist of the conversion of a Term into a BoolExpr n.

All these pieces are assembled in the proveTautology function.

```
proveTautology : (t    : Term) →
                 {pf : isSoExprQ (stripPi t)} →
                 let n = freeVars t in
                   {pf2 : isBoolExprQ n (stripPi t) pf} →
                   let b = concrete2abstract t n {pf} {pf2} in
                     {i : foralls b} →
                     proofGoal 0 n (zeroleast 0 n) b []
proveTautology t {_} {_} {i} =
  soundness (concrete2abstract t (freeVars t)) i
```

The proveTautology function converts a raw Term to a BoolExpr n format and calls the soundness lemma. It uses a few auxiliary functions such as freeVars, which counts the number of variables (needed to be able to instantiate the n in BoolExpr n), and stripSo & stripPi, which peel off the universal quantifiers and the function So with which we wrap our tautologies. These helper functions have been omitted for brevity, since they are rather cumbersome and add little to the understanding of the subject at hand.

These are all the ingredients required to automatically prove that formulae are tautologies. The following code illustrates the use of the proveTautology functions; we can omit the implicit arguments for the reasons outlined in Sec. 2.4.

```
exclMid : (b : Bool) → P (b ∨ ¬ b)
exclMid = quoteGoal e in proveTautology e

peirce  : (p q : Bool) → P (((p ⇒ q) ⇒ p) ⇒ p)
peirce  = quoteGoal e in proveTautology e

fave    : exampletheorem   -- defined in Fig. 4.3
fave    = quoteGoal e in proveTautology e
```

This shows that the reflection capabilities recently added to Agda are quite useful for automating certain tedious tasks, since we now need not encode the Boolean expression twice, in slightly different formats. The conversion now happens automatically, without loss of expressive power or general applicability of the proofs resulting from soundness. Furthermore, by using the proof by reflection technique, the proof is generated automatically.

It seems conceivable to imagine that in the future, using techniques such as those presented here, a framework for tactics might be within reach. Eventually we might be able to define an embedded language in Agda to inspect the shape of the proof that is needed, and look at a database of predefined proof recipes to see if one of them might discharge the obligation. An advantage of this approach versus the tactic language in Coq, would be that the language of the propositions and tactics is the same.

The attentive reader will remember that we previously studied a system capable of automatically quoting concrete Agda to a simple user-defined AST. Would that not be perfectly suited to quoting to the BoolExpr type used here? This turns out to be the case: we exploit this possibility in the rest of this chapter.

4.3.1 An Aside: Real-world Example of Automatic Quoting

The process of quoting to a BoolExpr n outlined in Sec. 4.3 quickly becomes an ugly mess, with functions checking properties of an expression (such as only certain functions like _∧_ or ¬_ occurring in the Term) being repetitive and verbose. The code summarised in Fig. 4.7 is an example of such a mess. If one then wanted to quote to some other AST, the whole process would have to be modified, which, I can guarantee, is a painful process.

The actual conversion function also ends up having many branches, checking if all the constructors and definitions are recognised, etc. This process can be made a lot less ugly and a lot more reusable. Recall the Autoquote module developed in Sec. 3.3; the same can be used here, both as an illustration of its use, and to avoid code duplication, thus making the code for term2boolexpr more concise.

Autoquote only supports simple inductive data types, so the first problem we encounter is that BoolExpr n has an argument of type Fin n to its constructor Atomic (see Fig. 4.1). To work around this, we introduce a simpler, intermediary data structure, to which we will convert from Term. This type, called BoolInter, is presented in Fig. 4.8. It has no such constraints.

```
data BoolInter : Set where
  Truth     :                                BoolInter
  Falsehood :                                BoolInter
  And       : BoolInter → BoolInter →        BoolInter
  Or        : BoolInter → BoolInter →        BoolInter
  Not       : BoolInter                →     BoolInter
  Imp       : BoolInter → BoolInter →        BoolInter
  Atomic    : ℕ                        →     BoolInter
```

Figure 4.8: An intermediary data type, which is a simplified (constraint-free) version of BoolExpr n.

The mapping needed for Autoquote is as follows: we mention which constructor represents De Bruijn-indexed variables and what the arity is of the different constructors. This way only Terms containing variables or the operators and, or, not, implication, true or false are accepted. Using this mapping, we can construct the function term2boolexpr that, for suitable Terms, gives us an expression in BoolInter. See Fig. 4.9.

Once we have a BoolInter expression, we just need to check that its variables are all in scope (this means that $\forall$ Atomic $x : x < n$, if we want to convert to a BoolExpr n). This is done in bool2fin, assuming that bool2finCheck holds (the latter simply expresses the in-scope property).

```
boolTable : Table BoolInter
boolTable = (Atomic,
    2 # (quote _∧_ ) ↦ And
  :: 2 # (quote _∨_ ) ↦ Or
  :: 1 # (quote ¬_  ) ↦ Not
  :: 0 # (quote true  ) ↦ Truth
  :: 0 # (quote false ) ↦ Falsehood
  :: 2 # (quote _⇒_) ↦ Imp :: [])
term2boolexpr : (t : Term)
              → {pf : convertManages boolTable t}
              → BoolInter
term2boolexpr t {pf} = doConvert boolTable t {pf}
```

Figure 4.9: The mapping table for quoting BoolInter.

```
bool2finCheck : (n : ℕ) → (t : BoolInter) → Set
bool2finCheck n Truth       = ⊤
bool2finCheck n (And t t₁)  = bool2finCheck n t × bool2finCheck n t₁
...
bool2finCheck n (Atomic x) with suc x ⩽? n
bool2finCheck n (Atomic x) | yes p  = ⊤
bool2finCheck n (Atomic x) | no ¬p  = ⊥
bool2fin : (n : ℕ) (t : BoolInter) (bool2finCheck n t) → BoolExpr n
bool2fin n Truth      pf       = Truth
bool2fin n (And t t₁) (p₁, p₂) = And (bool2fin n t p₁) (bool2fin n t₁ p₂)
...
bool2fin n (Atomic x) p₁ with suc x ⩽? n
bool2fin n (Atomic x) p₁ | yes p = Atomic (fromℕ⩽ {x} p)
bool2fin n (Atomic x) ()  | no ¬p
```

With these ingredients, our concrete2abstract function presented in Sec. 4.3 can be rewritten to the following drop-in replacement, illustrating how useful such an abstraction can be. It uses the function term2boolexpr defined in Fig. 4.9.

```
concrete2abstract' :
  (t : Term)
  → {pf : isSoExprQ (stripPi t)}
  → let t' = stripSo (stripPi t) pf in
    {pf2 : convertManages boolTable t'}
    → (bool2finCheck (freeVars t) (term2boolexpr t' {pf2}))
    → BoolExpr (freeVars t)
concrete2abstract' t {pf} {pf2} fin = bool2fin (freeVars t)
                                               (term2boolexpr
                                                 (stripSo (stripPi t) pf)
                                                 {pf2})
                                               fin
```

Clearly, the Autoquote module can save a lot of repetitive coding for converting Terms into some more structured AST, such as BoolExpr n.

Finally, all developments regarding the proof by reflection technique, including the quoting code can be found in the modules Proofs.TautologyProver and Metaprogramming.Autoquote, respectively. There are also examples of using the tautology prover as a library in Proofs.ExampleTautologies, as well as two examples of using Autoquote in Metaprogramming.ExampleAutoquote. The more extensive illustration of what is possible using Autoquote can be found in Proofs.TautologyProver.

Chapter 5

Type-safe Metaprogramming

Another area in which an application for the new reflection API was found is that of type-safe metaprogramming, taking advantage of Agda's powerful type system.

Metaprogramming is a technique which is already widely used, for example in the Lisp community, and involves converting terms in the concrete syntax of a programming language into an abstract syntax tree that can be inspected and/or manipulated, and possibly be made concrete again. Afterwards it can be evaluated as if it were code the programmer had written directly. In Agda the reflection happens at compile time, allowing for the strong static typing we have come to know and love. If run time reflection were possible, any program compiled with Agda would need to include the complete typing system, a problem which does not exist in Lisp, since it is dynamically typed, which makes run time reflection possible. In Agda, therefore, a compromise of sorts is required.

Reflection is well-supported and widely used in Lisp and more recently in Haskell, using the Template Haskell compiler extension [51]. It has enabled much automation of tasks otherwise requiring *boilerplate*[1] code, such as generating embedding-projection function pairs for generic programming. One such example is due to Norell and Jansson [41].

Clearly, metaprogramming with Template Haskell is a very useful technique, but it does have a conspicuous cumbersomeness (or should we say, potential pitfall). Developing a piece of Template Haskell code which should generate some function often results in debugging type errors in the resulting machine-generated code. This is a tedious and painful process, since, typically, generated code is much less self-explanatory or readable than hand-written code.

Here we propose a new way of looking at metaprogramming, namely type-safe metaprogramming. It would be great to be able to define some data structure for, say, lambda calculus, and have the guarantee that any term constructed in this AST is type-correct. The obvious advantage is then that the type checker will catch errors in whichever method tries to build an invalid piece of abstract syntax at compile time. This is preferable to the type checker giving an obscure error pointing at some generated code, leaving the programmer to figure out how to solve the problem.

[1] According to the Oxford English Dictionary, boilerplate is defined as *"standardised pieces of text for use as clauses in contracts or as part of a computer program."*

In this chapter we will explore how one can leverage the power of dependent types to achieve more type safety when writing metaprograms.

5.1 Preamble

In this section about metaprogramming, the object language we will be studying is the simply typed lambda calculus (STLC). Although the reader is assumed to be familiar with the rules and behaviour of STLC, the definitions and rules which will be relevant later on are briefly repeated here.

We first introduce the idea of contexts. A context is a stack of types, in which one can look up what type a variable is supposed to have. We have empty contexts, $[]$, and the possibility of adding a new type to the top of the context stack. We denote extension of the context by $\_::\_$, so x :: xs means x pushed on the context xs. There are also typing assumptions, of the form $x : \sigma$. This means the variable x has type σ. We also introduce the notion of a typing relation, or judgement, $\Gamma \vdash t : \sigma$, meaning that given some context Γ, the term t has type σ.

The typing rules are written using horizontal bars. Above the bar are the assumptions, and below the bar are conclusions we may draw if those assumptions hold. The validity of a typing judgement is shown by providing a typing derivation, constructed using the typing rules. See Fig. 5.1 for the typing rules.

$$[\text{var}]\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad [\text{lit}]\frac{c \text{ constant of type } \sigma}{\Gamma \vdash c : \sigma}$$

$$[\text{lam}]\frac{x : \sigma :: \Gamma \vdash e : \tau}{\Gamma \vdash (\lambda x : \sigma . e) : \sigma \to \tau} \quad [\text{app}]\frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau}$$

Figure 5.1: The typing rules for simply typed lambda calculus.

Of special interest are terms which we call *closed*. Closed is defined as being typable under the empty context, $[]$. Such terms do not refer to variables which were not introduced by lambda abstractions in that same term (a.k.a. terms free of free variables), and are also sometimes referred to as *combinators*.

Here we have used named variables, but in the following section these will be replaced in favour of De Bruijn indices.

5.1.1 De Bruijn Indices

Since we assume familiarity with lambda calculus in general, only a short introduction will be given here regarding nameless De Bruijn-indexed lambda terms [17], as opposed to the "usual" named representation which is surprisingly enough still the standard for most textbooks on the subject. Named representation of lambda terms has all sorts of intricate issues such as preventing capture of free variables after α-conversion, and needing to generate fresh variable names when adding abstractions, to name but a few difficulties. Algorithms for transforming and generating lambda terms are often riddled with "bookkeeping" to prevent such unwanted behaviour. For example, whole libraries [62] have been

developed to work out of the box and do these sort of operations generically. On the other hand, De Bruijn representation has the drawback that variable names are context sensitive. This discussion is, however tempting it may be to speak derisively about named lambda representation, rather outside the scope of this project, so we will restrict ourselves to a short presentation of the De Bruijn representation.

Usually, terms are denoted with a lambda abstraction binding a variable, and then later, in their bodies, referring to these bound values by name. Not so with De Bruijn indices, where a variable is represented by a natural number counting the number of abstractions between its occurrence and binding λ. To illustrate the concept we present some example terms in Table 5.1.

Named	De Bruijn
$\lambda x.x$	$\lambda.0$
$\lambda x.\lambda y.xy$	$\lambda.\lambda.1\ 0$

Table 5.1: A few sample translations from named lambda terms to De Bruijn-indexed terms.

Obviously, $\lambda y.y$ and $\lambda x.x$ are essentially the same lambda term, but represented differently. This is a "problem" we do not encounter using De Bruijn indices, since lambda expressions have one canonical representation. Also, because of the fact that a variable's index may not be higher than its depth, it is trivial to check that terms are closed, which makes the De Bruijn representation ideal for representing combinators. In all of the algorithms presented in this chapter, De Bruijn representation will be used.

5.1.2 Modelling Well-typed λ-calculus

For the running example in this section, we will look at simply typed lambda calculus (STLC) with the usual type and scoping rules, as defined in Fig. 5.2. All the modules that deal with lambda expressions (everything in the Metaprogramming name space of the project) work on this WT (which stands for well-typed) data type. Notice how the constructors are basically a transliteration of the STLC typing rules introduced in Fig. 5.1, save the addition of a size parameter.

The first thing to notice is that all terms in WT are annotated with a context, a type (the outer type of the lambda expression), and a size. The size is an arbitrary measure which should be strictly larger for terms which are structurally larger. This will become useful later, when we need to show that certain functions preserve the size of terms, but other than that the size has no interesting meaning. It is tempting to make the size parameter implicit, in the hope that this will hide a lot of clutter. Unfortunately, in most of the functions in this chapter, the size of terms needs to be specified to enable Agda to solve constraints between input and output of transformation functions. This is why the choice has been made to keep the size argument explicit, and occasionally use the underscore to tell Agda to infer the size when possible. After all, if we need to pattern match on implicit parameters and pass them along anyway, the clutter is only worse than if they were explicit.

The type annotations are elements of U′, defined in Fig. 5.3, which models base types and arrows. Contexts are simply lists of types, the position of

```
data WT : Ctx → U′ → ℕ → Set where
  Var  : ∀ {Γ} {τ}
       → τ ∈ Γ
       → WT Γ       τ          1
  Lam  : ∀ {Γ} σ {τ} {n}
       → WT (σ :: Γ) τ          n
       → WT Γ       (σ ⇒ τ)    (suc n)
  _⟨_⟩ : ∀ {Γ} {σ τ} {n m}
       → WT Γ       (σ ⇒ τ)    n
       → WT Γ       σ          m
       → WT Γ       τ          (suc n + m)
  Lit  : ∀ {Γ} {x}
       → Uel x
       → WT Γ       (O x)      1
```

Figure 5.2: The simply typed lambda calculus with De Bruijn indices.

elements of the list corresponding to their De Bruijn indices.

```
data U′ : Set where
  O     : U          → U′
  _⇒_   : U′ → U′    → U′
  Cont  : U′         → U′
Ctx : Set
Ctx = List U′
```

Figure 5.3: The universe used inside the metaprogramming libraries, with base and arrow types, parameterised by a user-defined universe U.

The O constructor, which stands for base types, is parameterised by an argument of type U. This is the user-defined universe by which all the library modules in Metaprogramming are parameterised. This would allow a user to instantiate the type checking module, Metaprogramming.TypeCheck, with a universe which has a representation of natural numbers, or Booleans, or both. Finally there is the Cont constructor, which will be used and explained later, in Sec. 5.3. In the following snippets of code we will present the other helper functions a user needs to define as we encounter them, summarising finally what is necessary and why in Sec. 5.5.

Finally there are the arguments of type $\tau \in \Gamma$ to the Var constructor. These are evidence that the variable identifier in question points to a valid entry of type τ in the context, Γ. Values of this type are basically annotated naturals corresponding to the De Bruijn index of the variable. This data type is defined in Fig. 5.4, and from such a value, one can query either the index (as a natural or Fin s, s being the length of the list) in the context (which is equal to their

De Bruijn index, given how entering the body of a lambda abstraction pushes a new entry onto the context) or the type of the variable they represent. Note that because of this the Var constructor is not parameterised with an explicit argument other than the _∈_ parameter.

```
data _∈_ {A : Set} (x : A) : List A → Set where
  here  : {xs : List A}                → x ∈ x :: xs
  there : {xs : List A} {y : A} → x ∈ xs → x ∈ y :: xs
```

Figure 5.4: The definition of the _∈_ data type, used as a witness that a variable with some type points to a valid location in the context. Note that _::_ binds more strongly than _∈_.

It should be clear that a term in WT [] is closed, since if the context of a term is empty and given that all WT terms are well-scoped, the only way to introduce variables (remembering that they require a proof of being in the context) is to first introduce an abstraction which extends the environment. This leads us to define the following alias for closed well-typed terms.

```
Well-typed-closed : U′ → ℕ → Set
Well-typed-closed = WT []
```

Looking at the WT data type constructor by constructor, we first encounter Var. The Var constructor stands for variables. A variable only has one argument, namely a proof that its index points to an entry in the context somewhere. Contexts are defined as lists of types, therefore τ is the type of the WT expression constructed by Var. Note that in particular, a variable cannot occur on its own within an empty context. There is no proof possible that a variable inhabits the empty context.

Next, we encounter abstractions, modelled by the Lam constructor. Here we are introducing a new variable with type σ into the context. Since we always push type variables on top of the context whenever we enter the body of a lambda abstraction, the index of the types in the context always corresponds to the De Bruijn-index of that variable. Intuitively, the deeper a variable in the list, the further away (in terms of lambda's) it is towards the outside of the expression, as seen from the point of view of the variable in question. Finally, a Lams second argument is its body, which is a well-typed term with type τ, given the abstraction's context extended with the type of the variable the lambda binds. This now produces a term of type $\sigma \Rightarrow \tau$, since we bind something of type σ and return something with the body's type.

The application constructor, _⟨_⟩, is next. It takes two arguments, namely well-typed terms which "fit" in terms of application. That is, if the second argument has type σ, then the first argument should have a type $\sigma \Rightarrow \tau$, for any τ. This application then produces a term of type τ.

There is also a Lit constructor, for introducing literal values (such as the number 5) into expressions. Among other things, this is useful for testing purposes.

We will explain the other elements present in Lit, such as the O-constructor and the Uel function, in Sec. 5.2.2.

Given these constructors, terms of type WT can only be instantiated if they are well-scoped, thanks to the proofs $\tau \in \Gamma$ in the variable constructors. They are also guaranteed to be well-typed, because all the terms are required to "fit" (for example in the outer types of lambda abstractions and applications).

5.2 Type Checking

Because it usually is impractical to require direct construction of WT terms, we would also like to offer a way of translating from some less constrained data type to WT, if possible. To this end, we introduce the data type Raw, given in Fig. 5.5, which is a model of lambda terms with De Bruijn indices that should look a lot more familiar to Haskell users, since most models of lambda expressions in Haskell-land are untyped (although this is possible using GADTs, see Cheney and Hinze [9]).

```
data Raw : Set where
  Var : ℕ                → Raw
  Lam : U′      → Raw    → Raw
  App : Raw     → Raw    → Raw
  Lit : (x : U) → Uel x  → Raw
```

Figure 5.5: The Raw data type, or a model of simply typed lambda expressions without any typing or scoping constraints.

We do include some typing information in Raws, but it is unverified. We require lambda terms and literals to be annotated with their type, because otherwise the type checker would become a type inferrer. While this is possible (Algorithm W [15] would suffice), it is a pain to implement in a language where only structural recursion is allowed by default. The difficulty is that the unification algorithm typically used with Algorithm W makes use of general recursion. This is in fact a topic of research in its own right, and therefore outside the scope of this project [37].

We choose instead to use the relatively straightforward, structurally recursive algorithm for type checking lambda terms presented in Norell's tutorial on Agda [43]. This algorithm was adapted from McBride's work in Epigram [38]. The function infer – defined in the following paragraph, incrementally – provides a view on Raw lambda terms showing whether they are well-typed or not. This view is aptly called Infer, and is defined in Fig. 5.6.

The Infer view makes use of the erase helper function, which returns the *erasure* of a WT term. The erasure of a typed term is simply the same term with typing information erased. Because erase is implemented exactly as expected, its definition is omitted.

The Infer view expresses that a term is either incorrectly typed using bad, which can be used on any term in Raw, or well-typed, which is shown using the ok constructor. The constructor ok also requires the corresponding witness in

```
erase  : ∀ {Γ τ n} → WT Γ τ n → Raw
data Infer (Γ : Ctx) : Raw → Set where
  ok   : {n : ℕ} (τ : U′) (t : WT Γ τ n) → Infer Γ (erase t)
  bad  : {e : Raw}                       → Infer Γ e
```

Figure 5.6: The view on Raw lambda terms denoting whether they are well-typed or not.

WT; we have already agreed that if a term can be represented in WT, it must be both well-scoped and well-typed. Moreover, this correspondence is enforced by defining the view on erase t, the erasure of t, as opposed to an arbitrary Raw term.

The infer algorithm, which provides the Infer view and therefore must generate WT terms corresponding to Raw terms, is presented here, in sections.

```
infer : (Γ : Ctx) (e : Raw) → Infer Γ e
infer Γ (Lit ty x) = ok (0 ty) (Lit {x = ty} x)
```

Of course, a literal on its own is always well-typed, and corresponds to a WT with whatever type the literal has. A variable is similarly easy to type check, except that it should not point outside the context. That is, it should have a De Bruijn index smaller than or equal to its depth. Here we look up the variable and return whatever type the context says it has, or, if it is out of scope, bad.

```
infer Γ (Var x)                 with Γ ! x
infer Γ (Var .(index p))        | inside σ p = ok σ (Var p)
infer Γ (Var .(length Γ + m))   | outside m = bad
```

Abstractions are well-typed if the body of the lambda is well-typed, under a context extended with the type of the variable the lambda binds. Indeed, binding a variable adds it to the context for the body of the abstraction, with index 0, since it is the "most recent" binding. This can be seen in the term $\sigma :: \Gamma$ in the Lam constructor of WT. The type of the abstraction is, as argued above, a function from the type of the binding to the type of the body, $\sigma \Rightarrow \tau$ here.

```
infer Γ (Lam σ e)          with infer (σ :: Γ) e
infer Γ (Lam σ .(erase t)) | ok τ t = ok (σ ⇒ τ) (Lam σ t)
infer Γ (Lam σ e)          | bad    = bad
```

The application case is the most verbose, since we need to check the type of the applicand (called e in the code), and assuming it has an arrow type (otherwise something is wrong), we then have to check that the argument (called e_1 in the code) has the same type as the left-hand side of the arrow. If all goes well, we are done.

```
infer Γ (App e e₁)          with infer Γ e
infer Γ (App .(erase t) e₁) | ok (Cont a) t   = bad
infer Γ (App .(erase t) e₁) | ok (O x) t      = bad
infer Γ (App .(erase t) e₁) | ok (τ ⇒ τ₁) t with infer Γ e₁
infer Γ (App .(erase t₁) .(erase t₂))
                            | ok (σ ⇒ τ) t₁
                            | ok σ' t₂        with σ =? σ'
infer Γ (App .(erase t₁) .(erase t₂))
                            | ok (.σ' ⇒ τ) t₁
                            | ok σ' t₂
                            | yes             = ok τ (t₁ ⟨ t₂ ⟩)
infer Γ (App .(erase t₁) .(erase t₂))
                            | ok (σ ⇒ τ) t₁
                            | ok σ' t₂
                            | no  = bad
infer Γ (App .(erase t) e₁) | ok (τ ⇒ τ₁) t
                            | bad = bad
infer Γ (App e e₁)          | bad = bad
```

The code which does all of this can be found in Metaprogramming.TypeCheck, the views and data type definitions are in Metaprogramming.Datatypes.

5.2.1 Quoting to Raw

It is a fine coincidence that the data type Raw closely matches the Term AST defined in the Agda compiler limited to lambda-related constructors, so it is relatively simple to massage the output of quoteTerm into an element of Raw, if it contains only a lambda expression. The code which does this (mostly the function term2raw) is to be found in Metaprogramming.TypeCheck. Since the conversion code is uninteresting and quite similar to the code presented in Sec. 3.3, it is omitted.

Since we have a conversion function from Term to Raw at our disposal, as well as a type checker, it is tempting to write something like typed1.

```
testgoal1 : Raw
testgoal1 = term2raw (quoteTerm λ (b : ℕ → ℕ) → (λ (x : ℕ) → b x))

typed1    : Well-typed-closed (typeOf testgoal1) _
typed1    = raw2wt testgoal1

seeTyped1 : typed1 ≡
  Lam (O Nat ⇒ O Nat)
      (Lam (O Nat)
           (Var (there here) ⟨ Var here ⟩))
seeTyped1 = refl
```

What we now have, is an automatic quoting of lambda terms into well-typed WT terms. Note that we are required to annotate the binders with types, because otherwise the quoteTerm keyword will return a lambda term with unknown as

the type annotation, which our type checker will not accept. In seeTyped1 we can inspect the resulting WT term.

5.2.2 Unquoting WT

Previously we saw how to quote lambda expressions to WT and then type check them. Conversely, we would also like to be able to construct a term in WT and use the unquote keyword to turn it back into concrete syntax, otherwise there would not be much practical use in being able to do transformations on WT terms.

The interpretation function for WT terms is mostly unsurprising; it must take a WT and return a Term, Agda's abstract representation, which we can then unquote. Note that we are discarding the type information; Term is represents untyped expressions.

The first few clauses are precisely what one would expect, except maybe for the Lit case. Here we see the first signs of the universe model which is implemented, namely a call to an undefined function quoteBack. The idea is that all the types in the U′ universe we are using (arrows and base types denoted with the constructor O) are parameterised by a user-defined universe U, which is used for the base types. We cannot know what types a user has modelled in their universe, so we have to require that they also provide a method which knows how to unquote values in their universe.

The value pleaseinfer is simply set to el unknown unknown, which means an unknown sort and unknown type. In this case, Agda will just infer the type before splicing the term into the concrete code. We know this will succeed, since WT terms are well-typed.

```
lam2term : {σ : U′} {Γ : Ctx} {n : ℕ} → WT Γ σ n → Term
lam2term (Lit {_} {σ} x) = quoteBack σ x
lam2term (Var x)         = var (index x) []
lam2term (Lam σ t)       = lam visible pleaseinfer (lam2term t)
```

The application case on the other hand is curious. Unfortunately this is motivated by practical limitations. The Term AST only allows introduction of applications with the var and def constructors, which stand for variables or definitions applied to a list of variables, respectively. Therefore, we use a function Apply, which just applies its first argument to its second, which gives us the possibility of introducing a def, giving it the actual application-arguments in the expected list-format.

```
lam2term (t₁ ⟨ t₂ ⟩) = def (quote Apply)
  (arg visible relevant (lam2term t₁) ::
   arg visible relevant (lam2term t₂) :: [])
```

We also would like to be able to recover the type of the term in concrete Agda. We first reconstruct a term of type Type, Agda's representation of types. These functions are also unsurprising: arrows are translated to arrows, and for base types we must once again invoke a user-defined function which can interpret

their universe values to Agda types. The Cont case should be ignored for now, since it has to do with the CPS transformation, which is introduced in Sec. 5.3.

```
el : U′ → Set
el (O x)       = Uel x
el (u ⇒ u₁)    = el u → el u₁
el (Cont t)    = ⊥
lam2type : {σ : U′} {Γ : Ctx} {n : ℕ} → WT Γ σ n → Set
lam2type {σ} t = el σ
```

Once we have these functions, it is easy to introduce a concrete function from a WT term as follows, using unquote and lam2term.

```
concrete :              lam2type typed1
concrete = unquote (lam2term typed1)
unittest : concrete ≡ λ (a : ℕ → ℕ) → λ (b : ℕ) → a b
unittest = refl
```

Note that the types are also preserved, since, even though we drop the annotations on the lambda terms when interpreting, we do give concrete a type signature which reflects the intended type of the lambda term. The unit test would have failed if we omitted the ℕ annotations on the variables, or changed them to another type.

It would be nice to also have included the call to unquote inside the definition of lam2term, which would result in a more concise definition of concrete, but unfortunately, unquote does its magic at compile time, and if it were used inside lam2term, then the value of its argument would not be known at compile time. This is why a user will have to use the unquote keyword explicitly each time a value (and not an argument) of type Term is to be cast back to concrete Agda.

5.3 Example: CPS Transformation

Given the fact that we can now easily move from the world of concrete Agda syntax to a well-typed lambda calculus and back, the obvious next step is to do something with these well-typed terms. Doing anything with these terms constitutes a program transformation, since lambda terms represent simple programs. An additional bonus feature we now have at our disposal is the ability to do these transformations while ensuring that certain properties (notably the well-typed property of our terms) are preserved.

The first case study in this area is that of transforming lambda terms into continuation-passing style (CPS). The idea of CPS is not new; it is what happens when one takes the primitive idea of computer programming, which essentially involves calling functions and returning values after their completion, and remove the notion of returning [5]. This seems both profound and unusable, since how will we get an answer from a function which is not allowed to return? Yet, it turns out to be a useful paradigm for many applications [33]: consider the example where one wants to print an integer, but before doing so, would like

to call, on that number, the function which increases integers by 1. That might look something like this fictional functional code.

```
main = print (suc 5)
```

If the idea of *returning* values is forbidden, how then must one use the result of suc? The answer is to do a transformation on the code; a continuation-passing style transformation. This name refers to the fact that functions which would normally do something analogue to issuing a `return` statement, are passed, as an additional parameter, a function to call on the result, instead of `return`.

The following translation provides an example.

```
factorial : ℕ → ℕ
factorial 0       = 1
factorial (suc n) = (suc n) * (factorial n)
factCPS : {a : Set} → ℕ → (ℕ → a) → a
factCPS 0       k = k 1
factCPS (suc n) k = factCPS n (λ f → mult f (suc n) k)
  where
    mult : {a : Set} → ℕ → ℕ → (ℕ → a) → a
    mult n m k = k (n * m)
```

Here we have translated the function by adding a new parameter (called k) which is called on the original result of the computation, instead of just returning that result. In the base case the translation was trivial, but in the inductive case we have to do a little more work. There, we immediately call the function recursively, asking for the factorial of the next smaller number, but providing a continuation which combines the result of that computation with the "current" value of n. We multiply the result of the recursive call with the current value of n by calling the mult function, which is also in continuation passing style; the result of this multiplication is what we would like to return, which is why we provide k as the continuation function.

We can now use this function in the traditional way if we pass the identity function as our continuation. This way, the result of the computation is returned unchanged. Notice, though, that the type of the CPS-transformed function is necessarily different from the original function.

Some anecdotal evidence that our CPS-transformed function does, indeed, perform as we expect it to is provided by equivFact1 and equivFact5.

```
equivFact1 : factorial 1 ≡ factCPS 1 id
equivFact1 = refl
equivFact5 : factorial 5 ≡ factCPS 5 id
equivFact5 = refl
```

This transformation can be done in a mechanical way, too. Also the type we expect the new function to have can be derived. This is discussed at length by Might [40], whose implementation was also used as inspiration for this type-

safe version. Weirich also presented a version of CPS in Agda during a course in 2009 [61], which was instructive during these developments.

Reynolds' overview [47] provides a good source of information on the history of the CPS transformation, which turns out to have been independently discovered in many fields. The CPS transformation of lambda terms was apparently first documented for Lisp programs by Fischer [20]. More references can be found in Danvy *et al.* [16], who present work on one-pass CPS transformations.

Pseudo code We will start by generalising the previous example, and giving an informal definition of the CPS transformation. The code in Fig. 5.7 is pseudo-Haskell, and should clarify the rough approach to doing a CPS transformation, before we add visual noise in the form of type preservation and termination guarantees.

```
M : Expr → Expr
M (Lam var expr) = Lam var (Lam k' (T expr k'))   -- with k' fresh
M expr           = expr
T : Expr → Expr → Expr
T (App f e) cont = T f (Lam f'                    -- with f' fresh
                        (T e (Lam e'              -- with e' fresh
                              (App (App f' e') cont))))
T expr      cont = App cont (M expr)
```

Figure 5.7: Pseudo-Haskell implementation of CPS transformation on a simple lambda language.

The function M adds a continuation parameter to lambda abstractions, and the T function takes an expression and continuation, and applies the continuation to the CPS-converted version of the expression.

In the function T, we see that applications are the only constructs which need real modification. Both applicand and argument (f and e here, respectively) are recursively CPS transformed, and the recursive call is given new lambda abstractions as the continuation term. These continuations simply bind the CPS-transformed f and e, called f' and e' here, and apply them as before, but now applying the cont continuation term, too.

To make things more transparent, this pseudo code still uses a named variable representation; the algorithm in this section will finally use the De Bruijn representation, as defined earlier in this chapter (see Sec. 5.1.2).

Type transformation Since we are mechanically transforming terms, we can predict the type of transformed terms from their original type. After fixing the types, we will be sure that the transformation is correct, since the resulting terms will have the expected type.

The type of a CPS-transformed function can be computed as follows, where RT stands for return type. This RT is some base type, i.e. O σ for some σ, and is a user-defined parameter to the module Metaprogramming.CPS (as well as to the

Datatypes module, but this parameter is automatically passed along to Datatypes in the CPS module). It denotes the desired return type from continuation functions.

Here we see the Cont constructor again. It is a tag we use to mark a type as going from some t ⇒ RT, where the cpsType function will be called recursively on t. Without this tag, it is difficult to keep track of which side of the function arrow to transform. It might be possible to get rid of this constructor, but so far I have not been able to do this. It is also a minor detail that does not make the application of reflection more or less relevant.

The types we get back from the cpsType function are to be interpreted as doing nothing in the base type case, since the CPS transformation of an atomic value will still be the atomic value, and in the arrow case, we transform the left of the arrow, then assume that the second argument will be a function from the original result type to our new result type, and finally dictate that the resulting function will also return a value in RT if given the correct first and second arguments.

```
cpsType : U′ → U′
cpsType (O x)     = O x
cpsType (t ⇒ t₁)  = cpsType t ⇒ (cpsType t₁ ⇒ RT) ⇒ RT
cpsType (Cont t)  = cpsType t ⇒ RT
```

The type we would like our transformation function to have is something which takes as input a term with some environment and type (WT Γ σ), a continuation (necessarily of type WT (map cpsType Γ) (cpsType (Cont σ)), namely an updated type context and a continuation-function for σ) and returns a semantically equal term with type WT (map cpsType Γ) RT, the return type. In other words, the continuation function must not rely on any variables which are not in the scope of the to-be-transformed function, and must produce a value of type RT. If these are then applied to each other, a value of type RT will be returned.

The algorithm We will now see the type-preserving algorithm. Below, it is presented case by case. It should be noted that this is still not the final version, since the WT type also has a size parameter. It is omitted here to keep clutter to a minimum; it will be added later, in Sec. 5.3.1.

```
T′ : {σ : U′} {Γ : Ctx} → WT Γ                    σ
                        → WT (map cpsType Γ) (cpsType (Cont σ))
                        → WT (map cpsType Γ) RT
```

The case for literals and variables is, as usual, not very difficult. All that happens here is that the continuation function is applied to the original term. The size arguments to WT have been omitted for brevity and the reader is assured that nothing exciting happens there.

Note that in the case of variables, some housekeeping needs to be done. We are actually changing all the values in the context (by applying cpsType to them), and we need to show that the same type, but CPS transformed, will be in the

same spot as the old type was. Therefore, a proof is given that if some variable with type σ is inside the environment Γ, then it will also be inside the new environment map cpsType Γ at the same index, but having value cpsType σ. The signature of cpsvar is given for reference; its proof is trivial.

```
cpsvar : ∀ {t g} → t ∈ g → cpsType t ∈ map cpsType g
T′ (Lit x)     cont = cont ⟨ Lit x ⟩
T′ (Var inpf) cont = cont ⟨ Var (cpsvar inpf) ⟩
```

The case for abstractions is slightly more involved: when T sees a lambda term, it adds a fresh continuation parameter, having type Cont t2, and then transforms the body of the lambda term into continuation passing style, asking it to invoke Var 0 on the result, which is the newly introduced continuation parameter. Variables are unchanged, except that their indices all need updating, since we have introduced a new lambda, so all the variables under that new lambda need an index-increase of 1. The function shift1 does this. This is a special case of a process called *weakening*, which refers to adding new variables to the context without changing the semantics of the term. Generalised weakening is implemented in a function called weak, which is excluded for brevity. This function adds a variable in an arbitrary position of the context, not just the top, as shift1 does, and adjusts the De Bruijn indices.

Note that even though we are introducing two abstractions, only one is new, since we are rebuilding the original lambda term but assigning the argument a new type, namely cpsType t1.

```
shift1 : ∀ {Γ τ n} → (τ₀ : U′) → WT Γ τ n → WT (τ₀ :: Γ) τ n
T′ {t1 ⇒ t2} (Lam .t1 expr) cont
  = cont ⟨ Lam (cpsType t1)
               (Lam (cpsType (Cont t2))
                    (T′ (shift1 (Cont t2) expr)
                        (Var here)
                    )
               )
         ⟩
```

Finally, we have the application case. Here, the values of both the applicand and the argument have to be converted into CPS.

The transform converts each with T′, and then catches their results in newly created continuations; note that both of the lambda abstractions are continuations.

```
T′ .{σ₂} {Γ} (_⟨_⟩ .{_} {σ₁} {σ₂} f e) cont =
  T′ f (Lam (cpsType (σ₁ ⇒ σ₂))
    (T′ (shift1 (σ₁ ⇒ σ₂) e) (Lam (cpsType σ₁)
      (Var (there here) ⟨ Var here ⟩
        ⟨ shift1 (cpsType σ₁) (shift1 (cpsType (σ₁ ⇒ σ₂)) cont) ⟩)))))
```

First f, the applicand, is transformed, with a new abstraction as the continuation. This abstraction must have a variable of the type of f, since it is the continuation which is to be invoked on f. The body of the abstraction is then the CPS transformation of e (after having shifted all the De Bruijn-indices up by 1 to compensate for the new abstraction), with again a continuation, this time binding a variable of the type of the argument (albeit transformed) and applying the transformed f (bound to Var 1) to the transformed e (here Var 0). Finally the original continuation, the one which was the argument called cont, is applied to the new f and e, but only after two shifts, resulting from the two lambda abstractions we introduced.

That wraps up the CPS algorithm. The full transformation algorithm can be seen in Metaprogramming.CPS, and examples of use, including a user-defined universe, are to be found in Metaprogramming.ExampleCPS.

5.3.1 Termination Bliss

Unfortunately, as the observant reader might have noticed, the algorithm T′ as presented in Sec. 5.3 is not structurally recursive, since in the recursive calls to T′ in the abstraction and application cases, we are applying shift1 to the constituent components of the input first. We can trivially see that the shift1 function does nothing to the size of the expression, but Agda's termination checker does not possess such intuition. As such, we will have to prove, by hand, that the algorithm is structurally recursive on its call graph.

Luckily, Bove and Capretta [6] come to the rescue by providing a recipe for this proof. Their method for mechanically taking a generally recursive algorithm and producing an auxiliary data type on which the algorithm *is* structurally recursive (the call graph, basically), which depends on a proof that the algorithm terminates on whatever input the user would like to call it on, is perfectly suited to this sort of situation. The curious reader is referred to Bove and Capretta's work for a thorough guide to this useful method.

After inspecting the recursive structure of the algorithm T′ we come to the conclusion that the data type TAcc presented below will do the job just fine.

```
data TAcc : {Γ : Ctx} {σ : U′} {n : ℕ} → WT Γ σ n → Set where
  TBaseLit : ∀ {Γ σ x} → TAcc (Lit {Γ} {σ} x)
  TBaseVar : ∀ {Γ σ x} → TAcc (Var {Γ} {σ} x)
  TLam     : ∀ {Γ t1 t2 n} {a : WT (t1 :: Γ) t2 n}
                     → TAcc (shift1 (Cont t2) a)
                     → TAcc {Γ} {t1 ⇒ t2} (Lam {Γ} t1 a)
  TApp     : ∀ {Γ σ σ₁ sza szb}
               {a : WT Γ (σ ⇒ σ₁) sza}
               {b : WT Γ σ szb}
                     → TAcc {Γ} {σ ⇒ σ₁} a
                     → TAcc (shift1 (σ ⇒ σ₁) b)
                     → TAcc (a ⟨ b ⟩)
```

In TAcc, each constructor of WT finds its analogue, and these proof terms are built having as arguments the proofs that TAcc can be constructed from the similar proofs on the arguments. Notice that the type TAcc has an index of type

WT, which is a term we promise the T′ algorithm will terminate on.

We can now add this TAcc argument to all the calls in T′, and Agda is now convinced the function terminates. All that is left is to prove that for all elements of wt ∈ WT we can construct a TAcc wt. The proof is as obvious as the data type was: we simply do recursion on the arguments of the constructors.

```
allTsAcc : ∀ {Γ σ n} → (wt : WT Γ σ n) → TAcc wt
allTsAcc (Var x)                    = TBaseVar
allTsAcc (Lit x₁)                   = TBaseLit
allTsAcc {_} {τ ⇒ σ} (Lam .τ wt)    =
  TLam (allTsAcc (shift1 (Cont σ) wt))
allTsAcc (_⟨_⟩ {Γ} {σ} {σ₁} wt wt₁) =
  TApp (allTsAcc wt)
       (allTsAcc (shift1 (σ ⇒ σ₁) wt₁))
```

But, horror! Agda now is suspicious that this function, allTsAcc, which is meant to give us the proof that T′ terminates given any WT term, does not terminate either! We also cannot apply Bove and Capretta's trick again, since by the construction of TAcc that would give us a data type isomorphic to TAcc.

Well-foundedness As it turns out, there is another trick up our sleeve: that of well-founded recursion. What we need to do is show that even though the recursion here is not structural, the terms do strictly decrease in size for some measure. Luckily we introduced a measure on WT long ago, the last argument of type ℕ. Following Mertens' example [22] we can build a well-foundedness proof for WT in terms of our measure, which we can then add as an extra argument to the allTsAcc function. The idea of proving well-foundedness in this fashion was first presented in Martin-Löf type theory by Paulson [45]; the implementations of the less-than ordering and inverse image relations in Agda's standard library, which we will use, are based on his work.

The first pitfall we encounter is that we want to define some Rel A which we will prove is well-founded on our data structure. The problem is that Rel is of type Set → Set₁ (not exactly, but for the purposes of argument), but WT is not of type Set, but Ctx → U′ → ℕ → Set. If we try to define something like λ {Γ σ n} → Rel (WT Γ σ n), things also become sticky rather quickly.

We can, however, circumvent this problem by defining a wrapper which is isomorphic to WT, but at the same time an element of Set. We will define this wrapper, WTwrap, as follows.

```
WTwrap : Set
WTwrap = Σ ℕ (λ n → Σ U′ (λ σ → Σ Ctx (λ Γ → WT Γ σ n)))
```

What is happening here is that we have defined a few nested dependent pairs, thus "hiding" the pi-type, which is what was causing us the headache. We will also need a function to inject WT into our wrapper type WTwrap, called to, but it is rather mundane. The function sz projects the size of the expression from WTwrap.

```
to : ∀ {Γ σ n} → WT Γ σ n → WTwrap
to {Γ} {σ} {n} wt = Γ , σ , n , wt
sz : WTwrap → ℕ
```

Now that we have this small bit of machinery, we can import the standard library's notion of well-foundedness and show that our measure, namely smaller than or equal to for WT elements, is well-founded.

We begin by showing that smaller-than is a well-founded relation on naturals.

```
<-ℕ-wf : Well-founded _<_
<-ℕ-wf x = acc (aux x)
  where
    aux : ∀ x y → y < x → Acc _<_ y
    aux zero     y  ()
    aux (suc x₁) .x₁ <-base     = <-ℕ-wf x₁
    aux (suc x₁) y  (<-step m) = aux x₁ y m
```

Now we use a lemma called inverse image from the Induction.WellFounded standard library module which shows that if we have some measure on a carrier, and a way to map some new type to this carrier type, we can lift the well-foundedness to the new type. We instantiate this lemma using our WTwrap wrapper, less-than on naturals, and a function sz which simply reads the size index which we already included in WT in Fig. 5.2.

```
module <-on-sz-Well-founded where
  open Inverse-image {_} {WTwrap} {ℕ} {_<_} sz public
  _≺_ : Rel WTwrap _
  x ≺ y = sz x < sz y
  wf : Well-founded _≺_
  wf = well-founded <-ℕ-wf
```

Next we must show that recursion on smaller or equal arguments is also fine, and that shifting the De Bruijn indices does not change the relative ordering of two WTpack elements (shift-pack-size). Note that weak is the generalised weakening function, which shift1 uses to add one type variable on top of the context stack and increase the De Bruijn indices by 1.

```
_≼_ : Rel WTwrap _
x ≼ y = sz x < (1 + sz y)
shift-pack-size : ∀ {τ Γ Γ' σ n} → (x : WT (Γ' ++ Γ) σ n)
                                 → to (weak {Γ'} {σ} {Γ} x τ) ≼ to x
shift-pack-size = ...
```

Note that for this to work, the natural number parameter to WT, which stands for a measure of expression size, is necessary, since if this was missing we would

have to define a fold on WT resulting in size instead of the simple projection the measure currently is, and my intuition says that this would make our well-foundedness proofs rather more involved (and possibly nonterminating too, bringing the problem full-circle). This is the motivation for adding such a $\mathbb{N}$ parameter to WT. It might be possible to eliminate this parameter from WT, but it was challenging enough to develop the well-foundedness proof as it is, so this further refinement is left as future work.

Once we have these ingredients, we can assemble it all to show that all calls to T′ with any WT terminate, and that the function allTsAcc also terminates. Our allTsAcc function now looks like this, showing only the "interesting" clauses.

```
allTsAcc : ∀ {Γ σ n} → (wt : WT Γ σ n)
                     → Acc _≺_ (to wt)
                     → TAcc wt
...
allTsAcc {_} {τ ⇒ σ} (Lam .τ wt)              (acc x) =
  TLam (allTsAcc (shift1 (Cont σ) wt)
         (x (to (shift1 (Cont σ) wt)) <-base))
allTsAcc (_⟨_⟩ {_} {σ} {σ₁} {n} {m} wt wt₁) (acc x) =
  TApp (allTsAcc wt
         (x (to wt) n<1+n+m))
       (allTsAcc (shift1 (σ ⇒ σ₁) wt₁)
         (x (to (shift1 (σ ⇒ σ₁) wt₁)) (n<1+m+n {_} {n})))
```

We now can export the final T′ translation function as T, so the user of the library need not worry about termination proofs. The function T′ terminates on all inputs anyway.

```
T : {σ : U′} {Γ : Ctx} {n m : ℕ}
  → (wt : WT Γ σ n)
  → WT (map cpsType Γ) (cpsType (Cont σ)) m
  → WT (map cpsType Γ) RT (sizeCPS n wt (allTsAcc wt (wf (to wt))) m)
T wt cont = T′ wt (allTsAcc wt (wf (to wt))) cont
```

The developments mentioned here, as well as termination proofs, can be found in Metaprogramming.CPS and Metaprogramming.WTWellFounded. Because terms tend to become pretty large, the examples are not shown here, but are presented in the module Metaprogramming.ExampleCPS.

Note that the final implementation of T now includes the size parameters on WT and the termination predicate defined here. As is suggested by all the auxiliary parameters to T, such as sized WT terms, termination predicates, etc. it was indeed less than trivial to get the CPS transformation working in a dependently typed setting. Although the development process was rather painful, we do now have a verified type-preserving CPS transformation.

5.4 Example: SKI Combinators

Another interesting application of our new type preserving program transformation framework is the proof of a rather old result in computer science, revisited. This result says that any closed lambda term (meaning being typable under the empty environment) can be translated to a simple combinatorial logic, having only a few primitives, and application. One such basis exists, using three combinators[2], $\{S, K, I\}$, as proven by Curry [14]. The 3 combinators of the SKI calculus are presented in Fig. 5.8. As shown by Fokker [21], it is possible to cut down this basis even further, resulting in a 1-element basis. Since this only makes the combinator terms more verbose, we will stick to the basis $\{S, K, I\}$.

```
s : ∀ {a b c : Set} → (a → b → c) → (a → b) → a → c
s = λ f → λ g → λ x → f x (g x)
k : ∀ {a b   : Set} → a → b → a
k = λ c → λ v       → c
i : ∀ {a     : Set} → a → a
i = λ x             → x
```

Figure 5.8: The three combinators which make up SKI combinator calculus.

Note that each of these 3 combinators are equivalent to closed lambda terms, but they form the basic building blocks of the SKI language. Basically, the SKI language is the same as the simply typed lambda calculus, except without the possibility of introducing new lambda abstractions, just the option to use one of these 3 predefined combinators. The fact that any closed lambda term can be translated to SKI may seem counter intuitive, but that is all the more reason to go ahead and, in the style of programs as proofs, prove that one can always translate a closed lambda term into SKI by defining this translation on the type Well-typed-closed. Because Agda is a sound proof assistant, we will have the guarantee that our function is total, and that the types of the terms are precisely preserved, which is a big advantage compared to the textbook implementations of SKI translation one finds written in Haskell, where there is nothing that says those functions cannot fail, except possibly a proof on paper. We prefer a machine-checked proof of the actual function at hand, since even if one has such a paper-proof, you have to trust that the semantics of the function on paper and the implementation are the same. Being used to programs as proofs tends to make you paranoid about using other programming paradigms.

Pseudo code We will first present and explain a pseudo-Haskell implementation of this translation; afterwards we will formalise it in Agda. The handwaving implementation is provided in Fig. 5.9.

[2] In fact, even I can be expressed in terms of S and K: $I \equiv S\ \langle\ K\ \rangle\ \langle\ x\ \rangle$, where the x may be an arbitrary combinator term, making the minimal basis S, K. This was noted by Schönfinkel [49].

```
compile : Lambda → Combinatory
compile (Var x)       = VarC x
compile (Apply t u)   = ApplyC   (compile t) (compile u)
compile (Lambda x t)  = lambda x (compile t)

lambda : String → Combinatory → Combinatory
lambda x t          | x ∉ vars t     = ApplyC K t
lambda x (VarC y)   | x ≡ y          = I
lambda x (ApplyC t u)                = ApplyC (ApplyC S
                                                (lambda x t))
                                              (lambda x u)
```

Figure 5.9: A pseudo-Haskell implementation of conversion from lambda terms to SKI calculus, using named variables.

Compared to the pseudo code implementation, we have the added complication that our WT type uses De Bruijn indices. This means that each time we replace a lambda abstraction with some other construction, we are potentially breaking the variable references, since some of them (exactly those in the body of the destroyed lambda) will need decrementing. Also, it sounds difficult to do a check on the variable's name to see if we should introduce an I or K in the variable case, but we will see that it is actually not so bad if we exploit the same context in the target language as in WT.

Formalisation We will first define a data type Comb in Fig. 5.10 which captures the SKI combinator language, extended with variables. One might be justified in starting to protest at this point, since we are introducing nonclosedness into the language, but notice that, in the same way as the WT type, we require variables to point to valid entries in the context, so that if we have a term of type Comb [], we know it contains no variables and thus is closed. We need these variables for intermediate results from the translation algorithm. This is also why we define the alias Combinator, which stands for a closed term in Comb, i.e. with an empty environment.

Note also that we have as much type safety in Comb as we have in WT, on account of the types of the arguments to the constructors needing to have sensible types. We could have chosen to use an untyped combinator language, and only do type checking after the translation is complete. In fact, type inference for SKI calculus has already been researched by Hindley [29]. The way we do it though, we are forced to have all intermediate terms preserve the types.

The translation of lambda terms into SKI presented in Fig. 5.11 is actually surprisingly (that is, if one is used to spending days grappling with the Agda compiler to get something seemingly trivial proven) straightforward. Since literals, variables and applications are supported, those can just be translated into the Comb equivalents without a problem, preserving the input context and type. The more complicated case occurs when we encounter a lambda abstraction.

If we were using named representation of STLC, we could write a function, called lambda, to be invoked with its corresponding variable name and the SKI-translated body, whenever we encountered an abstraction (our version of lambda

```
data Comb : Ctx → U′ → Set where
  Var  : ∀ {Γ} → (τ : U′) → τ ∈ Γ      → Comb Γ τ
  _⟨_⟩ : ∀ {Γ σ τ}
       → Comb Γ (σ ⇒ τ) → Comb Γ σ → Comb Γ τ
  S    : ∀ {Γ A B C}
       → Comb Γ ((A ⇒ B ⇒ C) ⇒ (A ⇒ B) ⇒ A ⇒ C)
  K    : ∀ {Γ A B}                     → Comb Γ (A ⇒ B ⇒ A)
  I    : ∀ {Γ A}                       → Comb Γ (A ⇒ A)
  Lit  : ∀ {Γ} {x} → Uel x             → Comb Γ (O x)
Combinator = Comb []
```

Figure 5.10: The data type Comb, modelling SKI combinator calculus. The Var constructor is less dangerous than it may seem.

is in Fig. 5.12). What we would like it to do is pattern match on this new translated body, and if it encounters a Var constructor, check if the variable has the same name. If it does, we evidently have encountered a $\lambda t.t$ somewhere in the expression, which should just translate to the I combinator. If the variable has another name, apply the variable to a K combinator, since we have encountered a $\lambda t.s$, and if s is just a variable, then it doesn't depend on the abstraction. In case we encounter an application as the body, we should recursively do the lambda-modification on the applicand and argument, then apply them both to the S combinator, since that will restore the analogue of the $\lambda x.\ s\ \langle\ t\ \rangle$ (bearing in mind that initially s and t might depend on x, being expressions and not necessarily atomic variables). Note that S ⟨ s ⟩ ⟨ t ⟩ indeed evaluates to λ f → λ g → λ x → f x (g x) applied to s then t, which gives λ x → s x (t x) which precisely reflects that we want s applied to t, and that they each might depend upon x.

```
compile : {Γ : Ctx} {τ : U′} {n : ℕ} → WT Γ τ n → Comb Γ τ
compile          (Lit x)       = Lit x
compile {_} {τ} (Var h)       = Var τ h
compile          (wt ⟨ wt₁ ⟩) = compile wt ⟨ compile wt₁ ⟩
compile          (Lam σ wt)   = lambda (compile wt)
```

Figure 5.11: The proof that any WT term can be translated into the Comb language.

In our version of lambda, in Fig. 5.12, we see that when we encounter a variable as the only thing in the body of the lambda, and if it is not the variable which is bound by the lambda under consideration, we decrement the De Bruijn index as promised, by peeling off a there constructor off the index-proof. If it is the variable bound by the lambda in question, we can replace the whole lambda expression with the identity combinator. Note also that the type of lambda is

Comb → Comb, so we will never encounter a Lam in its result.

```
lambda : {σ τ : U'} {Γ : Ctx} → Comb (σ :: Γ) τ
                              → Comb Γ (σ ⇒ τ)
lambda {σ}       (Var .σ here)          = I
lambda {σ} {τ}   (Var .τ (there i))     = K ⟨ Var τ i ⟩
lambda (t ⟨ t₁ ⟩) = let I1 = lambda t
                        I2 = lambda t₁
                    in                   S ⟨ I1 ⟩ ⟨ I2 ⟩
lambda           (Lit l)                = K ⟨ Lit l ⟩
lambda           S                      = K ⟨ S   ⟩
lambda           K                      = K ⟨ K   ⟩
lambda           I                      = K ⟨ I   ⟩
```

Figure 5.12: The function we invoke whenever we encounter a lambda abstraction during translation to SKI calculus.

It would not have been possible to define a total translation to SKI if the Comb data type did not have the same notion of variables and their restricted connection to contexts. Either that, or we would not have been able to guarantee that a closed lambda term induces a closed SKI term. Also, if names had been used to identify variables, one might have used the same mechanism of guaranteeing presence of the variables in the context, _∈_, but then an additional concept of uniqueness would have been necessary, both of which the De Bruijn representation provide for free. There also exist a few methods for directly translating from lambda terms to SKI combinators based only on De Bruijn variable identifiers [19], but apart from producing bloated SKI terms (since at least n K combinators are introduced if the variable's index is n – a sort of n-ary constant function is built up), implementing this algorithm in a well-typed setting is nearly impossible as a result of the fact that the intermediary terms returned by the recursive calls when abstractions or variables are encountered have radically different (although predictable) types. These reasons lead to the belief that the algorithm presented here is the most elegant of the options explored.

With this machinery in place, we can now successfully convert closed lambda expressions to SKI combinator calculus.

```
testTermWT : Well-typed-closed (typeOf (
  term2raw (quoteTerm λ (n : ℕ → ℕ) → λ (m : ℕ) → n m))) _
testTermWT = raw2wt (
  term2raw (quoteTerm λ (n : ℕ → ℕ) → λ (m : ℕ) → n m))

unitTest1 : compile testTermWT ≡
  S ⟨ S ⟨ K ⟨ S ⟩ ⟩ ⟨ S ⟨ K ⟨ K ⟩ ⟩ ⟨ I ⟩ ⟩ ⟩ ⟨ K ⟨ I ⟩ ⟩
unitTest1 = refl
```

Here we see how the existing lambda expression quoting system is used to read a concrete Agda lambda expression into a WT value, which is then compiled to produce an SKI term. The function unitTest1 displays what the end result is.

The resulting terms are sometimes rather unwieldy, as is illustrated in the examples provided in the module Metaprogramming.ExampleSKI, but this is to be expected, since the SKI calculus is obviously not very concise. If more readable terms are desired, one option to consider is adding extra combinators, called super-combinators, such as the o combinator, defined as follows [19].

```
o : ∀ {A B C} → Combinator ((B ⇒ C) ⇒ (A ⇒ B) ⇒ A ⇒ C)
o = S ⟨ K ⟨ S ⟩ ⟩ ⟨ K ⟩
```

Notice that the o super-combinator is really just function composition, as can be seen by the type signature. We take a function f and a function g as the first two arguments, then a value of type A, and then apply to this value f *after* g, precisely the definition of function composition, usually denoted _∘_.

Introducing such super-combinators could considerably shorten the representations of SKI terms, but being outside the scope of this example, we will stick with only the S, K and I previously defined. It is, however, interesting to note that because all lambda expressions can be translated to expressions using only S, K and I, these new super-combinators would simply be aliases for various combinations of the already-defined combinators.

5.4.1 From SKI to Concrete Agda

Once we have converted some lambda term to SKI, we might want to use it as a function on concrete Agda values. This is slightly pointless, since we already had some term to SKI-convert, so we might as well use that directly, but for completeness we do provide a translation from SKI back to WT, which we know we can unquote, as shown in Sec. 5.2.2.

Since the SKI combinators are themselves defined in terms of lambda expressions, it is trivial to first encode them as WT values (see Fig. 5.13), and then use those to assemble a traditional WT term from a value of type Comb. The unsurprising code, which is just a fold, can be found in Fig. 5.14.

```
Srep : ∀ {A B C Γ} → WT Γ ((A ⇒ B ⇒ C) ⇒ (A ⇒ B) ⇒ A ⇒ C) _
Srep {A} {B} {C} = Lam (A ⇒ B ⇒ C) (Lam (A ⇒ B) (Lam A
  (Var (there (there here)) ⟨ Var here ⟩ ⟨ Var (there here) ⟨ Var here ⟩ ⟩)))
Irep : ∀ {A Γ} → WT Γ (A ⇒ A) _
Irep {A} = Lam A (Var here)
Krep : ∀ {A B Γ} → WT Γ (A ⇒ B ⇒ A) _
Krep {A} {B} = Lam A (Lam B (Var (there here)))
```

Figure 5.13: The SKI combinators as represented in the WT data type.

Note that because WT is just as strictly typed as the Comb type, we are not losing any type safety on the way. The function combsz which can be seen in the ski2wt type signature simply calculates the natural representing the size of the final expression in WT. This is necessary because the value cannot be inferred.

```
ski2wt : {Γ : Ctx} {σ : U'} → (c : Comb Γ σ) → WT Γ σ (combsz c)
ski2wt (Var σ h)  = Var h
ski2wt (c ⟨ c₁ ⟩)  = ski2wt c ⟨ ski2wt c₁ ⟩
ski2wt S          = Srep
ski2wt K          = Krep
ski2wt I          = Irep
ski2wt (Lit x₁)   = Lit x₁
```

Figure 5.14: Translating SKI calculus back to lambda terms in the WT type.

In closing We have now defined a round-trip, automatic translation from concrete Agda lambda terms, to well-typed lambda terms in our WT representation, to SKI combinators as another data structure but preserving the type and scope guarantees provided by WT, back into concrete Agda terms, which are the semantic equivalent of the original terms.

By way of closing remarks, it is true to say that this chapter makes a persuasive argument to embrace the strong guarantees one can make using rich data types in a dependently typed language. Like many things in life, this advantage is something of a trade-off: the construction of a total, simply recursive algorithm which at the same time preserves types of terms *at every step* can be quite challenging. For example, the SKI algorithm shown here is pieced together from various sources, and initially, it seemed as if using De Bruijn representation was going to make things very complicated. The reason for this was that some implementations [19] simply remove all lambda abstractions, and when encountering a variable, use the index to determine how many K's should be used to produce a Frankensteinian indexing term. This is unfavourable for the large terms it generates, but also because the return type of the compile algorithm is, though predictable, very changeable. In the case of the CPS transformation, the type of the algorithm was less of a stumbling block, but as we saw in this chapter, termination posed rather a problem. This is something we take for granted when using dependent types to prove strong properties about our algorithms: the safety we get comes at the price of having to think a lot harder about each function, return type, clause, etc.

For those interested in looking at the full source in more detail, these developments can be found in the module Metaprogramming.SKI, and a few example translated terms as well as a guide to how to use the provided code as a library, reside in Metaprogramming.ExampleSKI.

5.5 Afterword: Parameters to Modules

As promised, we provide a summary of the parameters to the modules Datatypes, TypeCheck, SKI and CPS here, because these are designed to work with a user-defined universe. Aside from the universe, though, the user is also required to provide a few easy-to-define helper functions. These functions are necessary because invariably, they rely on pattern matching, which is something which is only possible if the to-be-used universe *and all of its constructors* are in scope.

The following list describes all the necessary parameters to the modules (note that not all modules require all parameters).

`U : Set`
: A data type representing the universe. It might have such elements as `Nat` and `Bl` which might stand for natural numbers and Boolean values.

`returnType : U`
: The return type for a CPS transformed function, detailed in Sec. 5.3.

`?type : U → Name`
: A function which, given an element of the universe, gives back the concrete Agda identifier which it stands for, such as `quote ℕ`.

`Uel : U → Set`
: An interpretation function, which returns the Agda type corresponding to some element of the universe.

`equal? : (x : U) → (y : U) → Equal? x y`
: A function which implements decidable equality between elements of the universe.

`type? : Name → Maybe U`
: A function which translates Agda identifiers into elements of the universe. Since failure is possible (the quoted term may be of some invalid shape), a `Maybe U` is expected.

`quoteBack : (x : U) → Uel x → Term`
: A function which can turn a value in the universe into an Agda `Term`.

`quoteVal : (x : U) → Term → Uel x`
: Finally, a function which, given an Agda term standing for a basic value, such as a natural, translates it into the universe.

An example of implementing such functions and instantiating the parameterised modules is found in Metaprogramming.ExampleUniverse. Defining an arbitrary universe should be straightforward after looking at that example.

Chapter 6

Generic Programming

Considering that Haskell and Agda, on the surface at least, seem like similar languages, it is not surprising that one of the inspirations for this project came from the (Template) Haskell world. As has been mentioned before, Template Haskell is a GHC compiler extension first described by Sheard and Peyton Jones [51]. It allows compile time metaprogramming, not unlike Agda's recent reflection API. One of the many useful applications of Template Haskell has been the automatic generation of embedding-projection function pairs for generic programming, saving on boilerplate code whenever a programmer wants to lift a new data structure to some generic universe [30]. Unfortunately, as we will see in Sec. 6.1, the reflection API in Agda is not yet powerful enough, in a number of ways, to be able to accomplish similar feats.

Another source of inspiration comes from Epigram, and McBride's idea of ornamentation of data structures [36]. This idea, which is to be implemented in a future version of Epigram, can be summarised as arguing that data type definitions are something one does not want to allow in a language, since they seem to drop out of the sky [39], and that actually, data type definitions should be no different from other value definitions. Ideally, there would only be one canonical data type definition, that can express all possible inductive data types; the **data** keyword we are used to would only be syntactic sugar for introducing a new value in this canonical data type. This way, reflection would not even be necessary, since, like in Lisp, data and functions are expressed in the same object language. Data type generic programming becomes normal programming. In work by Chapman, Dagand, McBride and Morris this idea is explored, hinting at implementation in Epigram 2 [8]. This idea has not been investigated in Agda, but some of the necessary components are available, which was another factor prompting the explorations detailed in this chapter.

6.1 Limitations

This section details what the limitations of the current reflection API are. Inspired by the ideas for automatic generic programming mentioned in the introduction to this chapter, we will try and see how far we get before running into trouble. Let us start with an example. Imagine a user has the following definition for a data type Col.

```
data Col : Set where
  Red Green Blue : Col
```

Obviously, this data type is isomorphic to Fin 3, the usual data type of natural numbers with an upper bound. It would be nice if we had a function which could, given the definition of Col, or at least a pointer to that definition, return the data type (if any) which is isomorphic to the user's type. For now we will assume we have such a function – we will call it isoDT. It is definable using the current reflection machinery, but because we do not use it, we will omit it as being an exercise to the reader to fill in. The idea would be to look at the list of constructors, and try and categorise them. If they have no arguments, then the type is simply an enumeration, which can be modelled using Fin. If they do have arguments, a sum-of-products representation could be generated, depending on how many there are and how many parameters they have.

```
isoDT : Name → Set
isoDT = { }0
```

The next logical move would be to write a function, which, given the pointer to Col's definition, and a value in Col, automatically returns the corresponding value in the isomorphic data type. This is possible, since we have shown in Sec. 3.2.1 that we can get a list of the constructors of a data type. At the very least, a naive implementation of this function, which we will call to, could return the element in Fin n which corresponds to the index of the given constructor in the list of constructors. Note that this would only work for trivial enumeration data types without parameters or indices. However, even this simple idea quickly gets stuck. Let us try and write down a type signature for the to function.

What we want is, given the Name of a type obtained with quote Col, a function from that type to the generic type which is isomorphic to it. This means that to (quote Col) yields a function with type Col → Fin 3, assuming that Fin 3 is the isomorphic generic counterpart to Col.

Our first attempt might be as follows.

```
to : (n : Name) → unquote (def n []) → isoDT n
```

The problem here, though, is that even though Col is indeed a definition taking no arguments, we cannot unquote def n [], since at compile time n is unknown, or as the Agda compiler aptly puts it, `n not a literal qname value`.

Another attempt might be the following, where we are not unquoting things at compile time, but rather ask the user to provide both the reference to the data type in question and its concrete Agda representation.

```
to : (n : Name) → (s : Set) → quote s ≡ def n [] → s → isoDT n
```

Here we run into another problem: we are not allowed to call quote s, since at compile time s is not a defined name, but some argument. A final attempt

seems to work a little better, and at least compiles, although we are clutching at straws.

```
to : (n : Name) → (s : Set) → quoteTerm s ≡ def n [] → s → isoDT n
to nm s pf x = { }0

testValue = to (quote Col) Col { }1
```

Here, the problem is that quoteTerm does manage at compile time, but produces a useless term, for the same reason that the last attempt failed. It returns var 0 [], in other words a reference to the nearest-bound variable, which results in a proof obligation var 0 [] ≡ def Col [] in hole 1 which we obviously cannot fulfil.

A similar problem arises if we want to be able to ask for the list of constructors of some type which is passed into a function as a parameter.

```
cs : (A : Set) → List Name
cs type = ... quoteTerm type ...
```

This causes the same problem as the previous snippet, where hole 1 was impossible, since the result from quoteTerm is simply var n [], for some n. What would be more useful, is if the result were a Name, such as Col, assuming that were the original parameter to cs. The call to quote would also not work here, because where it is used, type is not a defined identifier, but a variable, and quote can only handle definitions.

We are now forced to conclude that, even though certain elements necessary for generation of embedding-projection functions are attainable, we are blocked relatively early in the development process by such minor issues as arguments to quoting functions having to be known at compile time, which almost immediately precludes generic functions parameterised by a data type. We might work around this by making the process of building embedding functions more interactive. For example, we could let the user ask for the data type isomorphic to theirs, then write down the type signature manually, thus sidestepping the unquote at compile time problem. Unfortunately, we would still run into the same problem later on, if we wanted to make a projection function.

Another problem is that the fact that quote and cohorts are implemented as keywords, causing a problem with abstraction in general, because something like map quoteTerm is impossible. Enabling this would make the reflection system quite a bit more powerful, since currently the reflection system is only two-stage [50]. We have programs and metaprograms, but no way of writing metaprograms resulting in metaprograms. This would require being able to quote the quoting keywords. Maybe the Term structure should be expanded with another constructor, keyword, although a cleaner solution can probably be devised.

Probably all these minor issues could be worked out so that automatic generic programming becomes possible, but the expectation is that this will require some changes to the reflection API. Possibly a future version of Agda will support this.

The other motivation for looking at Agda from a generic programming per-

spective was the data type of data types idea mentioned earlier [8]. It would be rather exciting if we could use the reflection API to automatically convert data type definitions which already had been declared by the user, to values of this data type of data types. The expectation is that this should be possible, since we can easily inspect the constructors of data types, and that the use of `unquote` should be limited, since the type-of-types values are just Agda values. If one would like to have embedding and projection pairs, however, the same problem outlined previously would arise: unquoting is not flexible enough. Because of this and a lack of time, no further research was done to ascertain whether this is, in fact, feasible.

Because the findings in this chapter were negative, and no usable pieces of code were developed, the source distribution does not include any code related to generic programming.

Chapter 7

Discussion

This project's main innovations are novel combinations of existing techniques; as a result, quite a number of subjects are relevant to mention here.

As far as reflection in general goes, Demers and Malenfant [18] wrote an informative historical overview on the topic. What we are referring to as reflection dates back to work by Brian Smith [52] and was initially presented in the Lisp family of languages in the 80s. Since then, many developments in the functional, logic as well as object-oriented programming worlds have been inspired – systems with varying power and scope.

People sometimes jokingly say that the more advanced a given programming language becomes, the more it converges towards Lisp [25], and that the more complex some piece of software becomes, the higher the likelihood of discovering somewhere in the source a badly defined, ad hoc implementation of a Lisp interpreter. The fact is, though, that it is becoming increasingly common to generate pieces of code from a general recipe, possibly giving rise to a more efficient specific implementation, or at the very least not having to reinvent the wheel. Reflection is becoming more common, to various extents, in industry-standard languages such as Java, Objective-C, as well as theoretically more interesting languages, such as Haskell [55]. Smalltalk, an early object-oriented programming language with advanced reflective features [23], is the predecessor of Objective-C. As such, it is surprising that industry programming does not use more of these advanced reflective features which have already been around for a long time.

This would seem to be the inspiration for the current reflection system recently introduced in Agda, although we shall see that it is lacking in a number of fundamental capabilities. If we look at the taxonomy of reflective systems in programming language technology written up by Sheard [50] we see that we can make a few rough judgements about the metaprogramming facilities Agda currently supports[1].

- Agda's current reflection API leans more towards analysis than generation,

[1] Of course, having been implemented during a single Agda Implementors' Meeting [4], the current implementation is more a proof-of-concept, and is still far from being considered finished, so it would be unfair to judge the current implementation all too harshly. In fact, I hope that this work might motivate the Agda developers to include some more features, to make the system truly useful.

- it supports encoding of terms in an algebraic data type (as opposed to a string, for example),
- it involves manual staging annotations (by using keywords such as quote and unquote),
- it is neither strictly static nor run time, but compile time. It behaves much like a static system (one which produces an object program, as does for example YAcc [32]) would, but does not produce intermediate code which might be modified later by the user. Note that this fact is essential for Agda to remain sound.
- It is homogeneous, because a representation of the object language lives inside the metalanguage (as a native data type),
- it is only two-stage: we cannot as yet produce an object program which is itself a metaprogram. This is because we rely on built in keywords such as quote, which cannot themselves be represented.

Other related work includes the large body of publications in the domain of data type generic programming [48, 36], where we found the inspiration to try and implement prior techniques in a dependently typed setting. Especially work by McBride, *et al.* involving ornamentation and levitation [8] is intriguing, and something which would have been very interesting to do is to embed the data type of data types in Agda and automatically convert existing **data** declarations (which we can inspect) into values of this type. This whole step would be unnecessary in a language which supports this *data type of data types* a priori, so that the conversion to and from this type would be unnecessary, and data type generic programming becomes normal programming.

Program transformations and their correctness (by various definitions) have long been a subject of research [44], and given more advanced languages with more powerful generative programming techniques, this will likely prove a continuing trend. For example, Guillemette and Monnier have researched various type preserving transformations in Haskell, using GADTs [26, 27]. This work has even led to a type preserving compiler for System F in Haskell, where the GHC type checker mechanically verifies that each phase of the compiler preserves types properly [28]. Type preserving CPS transformations have also been studied, for example in Watanabe's thesis [60]. His work presents, among other things, a type preserving CPS transformation of De Bruijn-style lambda calculus, implemented in Coq.

As such, the contribution made in this project of a type-safe and total translation of simply typed lambda calculus to a language of SKI combinator calculus, as well as the continuation-passing style transformation, are interesting case studies. We have shown that these translations are usable in combination with a reflective language, making the process of translation of programs straightforward for users. Possible future work includes extending the body of available translations using the well-typed model of lambda calculus presented here as an intermediary language (or at least as inspiration for some other, more specialised data structure). It might also serve as a motivation to make the unquote keyword type-aware. Currently, even if all the steps in a transformation are type-safe, at the last step the typing information is still thrown away, which seems

like a wasted opportunity. Probably it would be easy to make unquote aware of the expected type, thereby making the final link in the program transformation framework type-safe.

As far as the proof techniques used in the section on proof by reflection (Chapter 4) are concerned, Chlipala's work [10] proved an invaluable resource, both for inspiration and guidance. One motivating example for doing this in Agda was Wojciech Jedynak's ring solver [31], which is the first example of Agda's reflection API in use that came to our attention. Compared to Jedynak's work, the proof generator presented here is more refined in terms of the interface presented to the user. The expectation is that approaches like these will become more commonplace for proving mundane lemmas in large proofs. The comparison to tactics in a language like Coq is a natural one, and we see both advantages and disadvantages of each style. Of course, the tactic language in Coq is much more specialised and sophisticated when it comes to generating proofs, but it is a pity that there are two separate languages in one, instead of the way it is in Agda, where metaprograms are written directly in the object language. Also, the fact that proof generation in Agda is explicit may be something some people appreciate. (Far) future work might be to implement some sort of tactic framework for Agda, possibly with a DSL in the style of Coq's tactic language, around the reflection API. The Ssreflect extension for Coq [24] should also be mentioned here; because of a lack of experience with Ssreflect, I refrain from making concrete statements, but the expectation is that the developments presented here should also be possible using Ssreflect.

Returning to our research question, repeated here to jog the memory, a summary of findings is made.

> "What are interesting applications of the new reflection API? Which tedious tasks can we automate? What advantages does the combination of dependent types and reflection give us? Finally, is the reflection API adequate as it stands to facilitate our needs or does it require extension? If extension is necessary, what kind and how much?"

This paper has presented two simple applications of proof by reflection, the latter using Agda's reflection API. Also, type-safe metaprogramming techniques have been demonstrated, offering automatic conversion and translation of programs, while preserving typing safety along the way. We have managed to automate generation of a certain class of proofs, which certainly would count as mundane. The clear advantage of Agda's reflection system is that it leverages the power of Agda's dependent types, leading to, among other yet to be described methods, the technique of type-safe metaprogramming presented here. Unfortunately, though, the reflection API itself is still rather primitive, so we find ourselves unable to define things such as an automatic Bove-Capretta transformation of a given function, or the generation of generic programming embedding and projection functions. The reasons for not being able to do all that we would like with the API as it stands are best summarised as follows.

- One cannot call unquote on nonconstructor terms, i.e. unquote (lam2term t) where t is some parameter or variable.
- It is impossible to introduce definitions, and therefore also impossible to

define pattern matching, since pattern matching is only allowed in definitions. Pattern matching lambda expressions in Agda are simply syntactic sugar for local definitions. This precludes automating the Bove-Capretta method, and makes generic programming techniques all the more painful.

- Inspection of functions (e.g. clauses) is not implemented, although inspection of data type definitions is quite comprehensive.
- By default, untyped terms are returned from the quoteTerm keyword. This has been solved in the patches presented in Appendix A.1, but these are yet to be included in the main development version of Agda.
- The unquote keyword is unaware of types, so even if a program transformation is type-safe, in the end unquoting is still hit-and-miss.

Having said all of that, though, a number of things are possible with the reflection mechanism as it stands, and the expectation is that it should be possible to define quite a few more examples of program transformations and proof generators which will likely turn out to be useful for various niche applications.

Acknowledgements

Obviously, a formidable number of people deserve thanks here, but I will refrain from mentioning everyone. Foremost, I would like to thank Wouter, my supervisor, for his infinite patience in explaining things, giving sound and complete advice, and his generally pleasant way of doing things. Marleen bravely proofread this work, gave much-needed moral support, was long-suffering: much appreciated. Tim deserves ample thanks for noticing overworkedness and nipping it in the bud, taking me on an epic hike through the forest. Justin did his bit by convincing me to go hitchhiking, which was surprisingly inspiring – a portion of this thesis was eventually written in a foreign city. The Friday Pie Day group is of course also worthy of mention, if only because of the added motivation I felt near the end of my project to catch up on all the wasted time spent drinking coffee and consuming calorific treats.

The rest of you know who you are; tolerating an atypically stressed-out me. Thanks.

Appendix A

Modifications to the Agda Compiler

During the course of this project, a few modifications were made to the Agda code base, to facilitate various processes. Since these modifications have not yet been included in the main code repository, anyone interested in trying out the changes is invited to make a clone of the forked repository where the development was done.

The compiler can be found at `https://darcs.denknerd.org/Agda`, and the modified standard library (modified to work with the updated data types in the compiler) can be found at `https://darcs.denknerd.org/agda-stdlib`. The instructions for installation of Agda from source, on the Agda wiki [2], can be followed unmodified. The modifications made are the following.

- The output of the reflection system (in other words the Term data type) was modified to include type annotations on lambda abstractions. See Sec. A.1.
- The compiler was extended to output a list of formatting rules based on the identifiers currently in scope. This is useful for producing syntax-highlighted documents from Literate Agda. See Sec. A.2.

A.1 Annotating Lambda Abstractions with Type

As mentioned in Sec. 3.1 it was necessary to slightly modify the representation of Terms that the reflection system returns to the user. What was needed was to annotate lambda abstractions with the type of their argument, since without this, type inferencing would be necessary. Even though this is possible, it would introduce unneeded complexity and open the can of worms that is type unification. As it turns out, the termination of type unification algorithms is something rather nontrivial to prove, even if solutions such as McBride's [37] do exist. To avoid this, the Term data structure internal to the Agda compiler was augmented with an optional field of type Type, which allowed two advantages. Firstly, it is now possible to distinguish between, for example, $\mathbb{N}$ and Bool variables in the same expression. Secondly, it allowed us to suffice with only

providing a type checker, as opposed to a full type inferencing function along with a type unifier, which poses a problem to the termination checker.

The changes required to the Agda compiler were rather small; the main thing that was needed was to extend the Term data type with a Maybe Type field to hold the extra parameter, and at most points where pattern matching on, or generation of such terms was done, an extra field needed to be added. Only the checkExpr function, which does type checking when a concrete Agda lambda term is encountered, needed to be adjusted, so that the inferred type of the argument to the lambda would be attached to the abstract syntax tree.

The actual code changes can be browsed on `https://darcs.denknerd.org`[1], but are not included here for brevity. You can also clone the complete modified compiler fork from there.

A.2 Automated Highlighting for Literate Agda

In the Emacs Agda mode, highlighting Agda source code currently only works after a module has been loaded, since then the rôle of various identifiers is known – be it constructor, function or type. Because of this, Löh's great LHS2TEX system [34] does not support automatic syntax highlighting of Agda code, but the documentation suggests using formatting rules, which are basically LATEX preprocessing macros. For example, `%format x = "\something{x}"`.

A small modification to the Agda compiler added an extension, available via the `--lagda` flag, which first loads the desired module, then if the module passes type checking, outputs a list of identifiers which are in scope, as a list of LHS2TEX format rules. The output of such a command, invoked using the usual parameters plus the `--lagda` flag can be piped into some file and then included in the main `lagda` file, as is done for this report. The user is expected to define a number of LATEX commands, though, which specify how the various source code tokens are to be formatted. The required commands are:

\defin the formatting for a definition like a function name,

\id the formatting for an identifier,

\fld the formatting for a field name, such as proj_1,

\con, \consym, \consymop formatting of a constructor or constructor operator, such as suc or _, _, and

\ty the formatting rule for a type.

Once again, the actual code changes can be browsed on `https://darcs.denknerd.org`[2]. Examples of using this system are to be found in the code for this paper: the Makefile specifies how to generate the formatting rules, and the main LATEX file shows how they are used.

[1] The following patches are interesting as far as typed lambda expressions go: from 20120724095751-a1717-7409480a0680c0e9b220070a0265970cb403c87e.gz to 20120802164956-a1717-213a839b6a17498d7fb0da67ea64c9603ca5409c.gz.

[2] The patches 20120621153102-a1717-bcec6bef23583acfb7fd06e3291a57e90d1b4c0b.gz to 20120625101400-a1717-6363a79683af6ad0752729ee24250e87d7af066b.gz are interesting as far as highlighting goes.

Appendix B

Guide to Source Code

This project is currently hosted at GitHub[1]. There you can find a few files containing the implementations of the presented algorithms, as well as the source for this paper, which is itself Literate Agda. Here a short summary is given of what each source file contains; see the directory tree presented in Fig. B.1 on page 76.

The `doc` directory contains the sources for this paper, which compile using the Emacs mode for Agda. The paper can also be generated again by running `make` in the `doc` directory. The file `ReflectionProofs.lagda` is the main LaTeX file used to generate this paper.

The `Metaprogramming` directory contains all the code relating to metaprogramming, namely the modules for CPS transformation (`CPS.agda`), SKI translation (`SKI.agda`), quoting (`Autoquote.agda`) and type checking (`TypeCheck.agda`), all in the appropriately named files. Examples of use for all the relevant modules are also provided, in the `Example...` modules. The `Util` folder contains a few helper functions. In `WTWellfounded.agda`, finally, the well-foundedness of the WT data type, under the natural measure, is proven.

The `Proofs` directory contains the proof by reflection experiments. The file `IsEven.agda` is where one can find the first example implementation of the even natural numbers proof generator, explained in Sec. 4.1. The file `TautologyProver.agda` implements the system described in Sec. 4.2 for quoting and proving Boolean tautologies. The `Util` folder contains some modules with boring lemmas and alias definitions.

[1] https://github.com/toothbrush/reflection-proofs

```
/
├── doc
│   └── ReflectionProofs.lagda
├── Metaprogramming
│   ├── Autoquote.agda
│   ├── CPS.agda
│   ├── Datatypes.agda
│   ├── ExampleAutoquote.agda
│   ├── ExampleCPS.agda
│   ├── ExampleSKI.agda
│   ├── ExampleTypeCheck.agda
│   ├── ExampleUniverse.agda
│   ├── SKI.agda
│   ├── TypeCheck.agda
│   ├── Util
│   │   ├── Apply.agda
│   │   ├── ConcreteSKI.agda
│   │   ├── Equal.agda
│   │   ├── ExampleShow.agda
│   │   └── PropEqNat.agda
│   └── WTWellfounded.agda
└── Proofs
    ├── ExampleTautologies.agda
    ├── IsEven.agda
    ├── TautologyProver.agda
    └── Util
        ├── Handy.agda
        ├── Lemmas.agda
        └── Types.agda
```

Figure B.1: Directory listing of the source distribution for this project.

Bibliography

[1] Agda developers. Agda 2.2.8 release notes. The Agda Wiki: http://wiki.portal.chalmers.se/agda/agda.php?n=Main.Version-2-2-8, 2012. [Online; accessed 6-April-2012].

[2] Agda developers. Agda installation instructions. The Agda Wiki: http://code.haskell.org/Agda/README, 2012. [Online; accessed 6-April-2012].

[3] A. Alexandrescu. *Modern C++ design*. Addison Wesley, 2001.

[4] Thorsten Altenkirch. [Agda mailing list] More powerful quoting and reflection? mailing list communication, https://lists.chalmers.se/pipermail/agda/2012/004127.html, 2012. [online; accessed 14-Sep-2012].

[5] K. Asai and O. Kiselyov. Introduction to programming with shift and reset. online, http://pllab.is.ocha.ac.jp/~asai/cw2011tutorial/main-e.pdf, 2011. [accessed 21-Aug-2012].

[6] A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

[7] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009.

[8] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 3–14, New York, NY, USA, 2010. ACM.

[9] James Cheney and Ralf Hinze. First-Class Phantom Types. Technical report, Cornell University, 2003.

[10] A. Chlipala. *Certified programming with dependent types*. MIT Press, 2011.

[11] Conor McBride. [Haskell-cafe] What's the motivation for η rules? message to Haskell-cafe mailing list, 30-Dec-2010, http://www.haskell.org/pipermail/haskell-cafe/2010-December/087850.html. [online, accessed 20-Aug-2010].

[12] Catarina Coquand, Dan Synek, and Makoto Takeyama. An Emacs interface for type directed support constructing proofs and programs. ENTCS, https://mailserver.di.unipi.it/ricerca/proceedings/ETAPS05/uitp/uitp_p05.pdf, 2006. [online; accessed 3-Sep-2012].

[13] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[14] H.B. Curry and R. Feys. *Combinatory logic*, volume 2. North-Holland, 1972.

[15] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[16] Olivier Danvy, Kevin Millikin, and Lasse R. Nielsen. On one-pass CPS transformations. *J. Funct. Program.*, 17(6):793–812, November 2007.

[17] N.G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[18] F.N. Demers and J. Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *Proceedings of the IJCAI*, volume 95, pages 29–38, 1995.

[19] Dan Doel. Haskell Hacking, Agda Enjoyment. online, http://code.haskell.org/~dolio/agda-share/html/SKI.html#131, 2012. [online; accessed 12-Sep-2012].

[20] Michael J. Fischer. Lambda calculus schemata. In *Proceedings of ACM conference on Proving assertions about programs*, pages 104–109, New York, NY, USA, 1972. ACM.

[21] Jeroen Fokker. The systematic construction of a one-combinator basis for lambda-terms. *Formal Asp. Comput.*, 4(6A):776–780, 1992.

[22] Eric Mertens (Galois). Introducing well-founded recursion. online, http://code.galois.com/talk/2010/10-06-mertens.pdf, 2010. [slides; accessed 31-Aug-2012].

[23] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[24] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010. RR-7392 RR-7392.

[25] Paul Graham. *Hackers and Painters: Big Ideas from the Computer Age*. O'Reilly, May 2004.

[26] Louis-Julien Guillemette and Stefan Monnier. A type-preserving closure conversion in Haskell. In Gabriele Keller, editor, *Haskell*, pages 83–92. ACM, 2007.

[27] Louis-Julien Guillemette and Stefan Monnier. Type-safe code transformations in Haskell. *Electronic Notes in Theoretical Computer Science*, 174(7):23 – 39, 2007. Proceedings of the Programming Languages meets Program Verification (PLPV 2006).

[28] Louis-Julien Guillemette and Stefan Monnier. A type-preserving compiler in Haskell. In James Hook and Peter Thiemann, editors, *ICFP*, pages 75–86. ACM, 2008.

[29] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:pp. 29–60, 1969.

[30] Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In Tarmo Uustalu, editor, *MPC*, volume 4014 of *Lecture Notes in Computer Science*, pages 209–234. Springer, 2006.

[31] Wojciech Jedynak. Agda ring solver using reflection. online, GitHub, https://github.com/wjzz/Agda-reflection-for-semiring-solver, 2012. [Online; accessed 26-June-2012].

[32] S.C. Johnson. *Yacc: Yet another compiler-compiler*. Bell Laboratories, Inc., 1975.

[33] Shriram Krishnamurthi. *Programming languages - application and interpretation*. e-book, 2003.

[34] Andres Löh. LHS2TEX, a preprocessor to generate LATEX code from Literate Haskell sources. online, http://www.andres-loeh.de/lhs2tex/, 2004. [online; accessed 11-Sep-2012].

[35] P. Martin-Löf. Constructive mathematics and computer programming. In *Proc. of a discussion meeting of the Royal Society of London on Mathematical logic and programming languages*, pages 167–184, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.

[36] C. McBride. Ornamental algebras, algebraic ornaments. *Journal of Functional Programming*, 2010.

[37] Conor McBride. First-order unification by structural recursion. *Journal of Functional Programming*, 13(6):1061–1075, 2003.

[38] Conor McBride. Epigram: practical programming with dependent types. In *Proceedings of the 5th international conference on Advanced Functional Programming*, AFP'04, pages 130–170, Berlin, Heidelberg, 2005. Springer-Verlag.

[39] Conor McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In Bruno C. d. S. Oliveira and Marcin Zalewski, editors, *ICFP-WGP*, pages 1–12. ACM, 2010.

[40] Matt Might. How to compile with continuations. http://matt.might.net/articles/cps-conversion/. [online, accessed 20-Aug-2012].

[41] U. Norell and P. Jansson. Prototyping generic programming in Template Haskell. In *Mathematics of Program Construction*, pages 314–333. Springer, 2004.

[42] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[43] Ulf Norell. Dependently typed programming in Agda. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, pages 1–2, New York, NY, USA, 2009. ACM.

[44] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Comput. Surv.*, 15(3):199–236, September 1983.

[45] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *J. Symb. Comput.*, 2(4):325–355, December 1986.

[46] K. Pitman. Special forms in Lisp. In *ACM Symposium on Lisp and Functional Programming*, 1980.

[47] John C. Reynolds. The discoveries of continuations. *Lisp Symb. Comput.*, 6(3-4):233–248, November 1993.

[48] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. *SIGPLAN Not.*, 44(2):111–122, September 2008.

[49] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92(3):305–316, 1924.

[50] Tim Sheard. Staged programming. online, http://web.cecs.pdx.edu/~sheard/staged.html. [accessed 20-Aug-2012].

[51] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, 2002.

[52] Brian Cantwell Smith. Reflection and semantics in LISP. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 23–35, New York, NY, USA, 1984. ACM.

[53] M.H. Sørensen and P. Urzyczyn. Lectures on the Curry-Howard isomorphism. 1998.

[54] Stackoverflow user maddy. "hey can't understand this piece of code" – a question about reflection. online, http://stackoverflow.com/questions/11965535/hey-cant-understand-this-piece-of-code. [accessed 20-Aug-2012].

[55] Aaron Stump. Directly reflective meta-programming. *Higher-Order and Symbolic Computation*, 22(2):115–144, 2009.

[56] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '97, 1997.

[57] A. S. Troelstra. From constructivism to computer science. *Theor. Comput. Sci.*, 211(1-2):233–252, 1999.

[58] T. van Noort, A. Rodriguez Yakushev, S. Holdermans, J. Jeuring, B. Heeren, and J.P. Magalhães. A lightweight approach to datatype-generic rewriting. *Journal of Functional Programming*, 20(3-4):375–413, 2010.

[59] P. Wadler. Proofs are programs: 19th century logic and 21st century computing. *Dr Dobb's Journal*, page 313, 2000.

[60] Yuki Watanabe. Study on proof of type preservation in CPS transformation. online, http://web.yl.is.s.u-tokyo.ac.jp/~ywtnb/master/thesis-submitted.pdf, 2011.

[61] Stephanie Weirich. Advanced Topics in Programming Languages: Dependent Type Systems. online, http://www.seas.upenn.edu/~sweirich/cis670/09/index.html, 2009. [online course material; accessed 23-Sep-2012].

[62] Stephanie Weirich, Brent A. Yorgey, and Tim Sheard. Binders unbound. *SIGPLAN Not.*, 46(9):333–345, September 2011.