École de technologie supérieure

Bachelor's Thesis

Formalizing *Types and Programming Languages* in Isabelle/HOL

Author:	Martin Desharnais
Professor:	David Labbé
Advisor:	Jasmin C. Blanchette
Advisor:	Dmitriy Traytel

December 15, 2014

Abstract

We formalize, using Isabelle/HOL, some languages present in the first two sections, namely “Untyped Systems” and “Simple Types”, of the book *Types and Programming Languages* [Pie02] by Benjamin C. Pierce. We first begin with a short tour of the λ-calculus, type systems and the Isabelle/HOL theorem prover before attacking the formalization *per se*. Starting with an arithmetic expressions language offering Booleans and natural numbers, we pursue, after a brief digression to de Bruijn indices, to the untyped λ-calculus. Then, we return to a typed variant of the arithmetic expression language before concluding with the simply typed λ-calculus.

Acknowledgements

First, I would like to thank Jasmin Blanchette, one of my advisors, who gave me the opportunity to move to Munich for an eight month internship as a research assistant. This experience gave me a taste of academic research and introduced me to the world of formal verification and theorem provers. I benefited greatly from the numerous discussions we had and the advices he gave me.

I am equally grateful to Dmitriy Traytel, my second advisor, for all the help he gave me for both my internship and my thesis. He taught me a lot and his help was invaluable for me to overcome some of the challenges I encountered.

I want to thank all members of the research chair for the warm welcome the gave me. A special thought goes to Lars Noschinski, my officemate, for all the conversations we had and all the insights he gave me of the German culture.

Contents

1 Introduction

This Bachelor's thesis deals with the formalisation of parts of the book *Types and Programming Languages* [Pie02], hereafter abbreviated *TAPL*, by Benjamin C. Pierce. The formalization is performed using the Isabelle/HOL theorem prover. This work concentrates on four languages, ranging from simple arithmetic expressions to the fully fledged λ-calculus, present in the first two sections, namely "Untyped Systems" and "Simple Types".

The main motivation to have chosen this subject is the intersection of personal interest and of opportunities provided by my internship at the chair for logic and verification at Technische Universität München. Having gradually developed an interest for programming languages in the last years, I was eager to learn more about the theory behind type systems. Pierce's book stood out as a reference recommended for a deep introduction to the main elements of this field. Also, as part of my internship, I worked on the implementation of the (co)datatype module in the Isabelle/HOL theorem prover. Having experienced the implementer role, I also wanted to learn about the user role and the process of formalization. Thus, the choice of this subject for this thesis was an opportunity to fulfill both goals.

Before entering into the realm of formalizations, we first introduce the required background (Section 2) for λ-calculus, type systems and Isabelle/HOL. The λ-calculus is a core calculus that captures the essential features of functional programming languages. That is, there exists a way to encode high level features such as recursion, datatypes, records, etc. Such calculus can come, as programming languages do, in two variants: typed and untyped. A type system is a syntactic method for automatically checking the absence of certain erroneous behaviors. To formalize these, we used Isabelle/HOL, an interactive theorem prover based on higher order logic. It resembles a functional programming language in that one can define datatypes and functions. The difference is that it allows to postulate properties of the formerly defined elements and to provide machine checked poofs that those properties are theorems.

The formalizations we perform all have a direct correspondence with chapters from *TAPL* (Section 3). We provide one Isabelle/HOL theory file per chapter.

The untyped arithmetic expressions language (Section 4) serves as a warm-up to experiment with the general structure of formalizations. It consists of Boolean expressions and natural numbers. This simplicity allows us to concentrate on the translation to Isabelle/HOL of the definitions found in the book and to accustom ourselves with the notation. Most of our definitions and theorems closely follows the ones from the book. The main exceptions are that we expose some hypotheses that are implicit in the book and that we use a different definition of the multi-step evaluation relation.

The formalization (Section 5) of the nameless representation of terms, also known as de Bruijn indices [Bru72], was not initially planned but arose from the need to use a concrete representation for variables in the λ-calculus. Our formalization closely follows the book.

While the previous formalizations are either a warm-up or a representation necessity, the untyped λ-calculus (Section 6) is the first core calculus we formalized. We differ from the book in the definition of the evaluation relation. In *TAPL*, the evaluation relation assumes that name clashes in variables are automatically solved by renaming them and, thus, ignore this possibility from there on. Such an assumption is not accepted by computer-verified proofs. We use de Bruijn indices as representation for variables to encode this assumption. Also, since the chapter in the book is more focused on explaining the λ-calculus, it contains no meaningful theorems. Nevertheless, we revisit the properties introduced with the arithmetic expressions language and either prove that they are still theorems or disprove them.

The typed arithmetic expressions language (Section 7) is again a warm-up; this time, to experiment with the formalization of a type system. Our formalization closely follows the book.

The simply typed λ-calculus (Section 8) is the second core calculus we formalize. Here, we differ significantly from the book, mainly because of our use of de Bruijn indices but also because of our representation of the typing context, we need to adapt some lemmas and replace others. This is certainly the most challenging part of the formalization, since we cannot follow the described proofs anymore and must find the right assumptions for our lemmas. In spite of these differences, our formalization still respects the spirit of the book since only the helper lemmas change, and the important theorems remain the same.

All sections combined, the formalization consists of 800 lines of definitions, theorems and exercises proposed in the book. It is publicly available[1] and can be executed with Isabelle 2014.[2] In this report, we focus on the definitions and how the theorems are expressed. When relevant, we present both the definitions from the book and our translation, highlighting and motivating the differences. Some proofs are presented but not explained. For a deeper insight into the proofs, the best methodology is to study the theory files in Isabelle.

[1]https://github.com/authchir/log792-type-systems-formalization

[2]http://isabelle.in.tum.de/website-Isabelle2014/

2 Background

2.1 Lambda-Calculus

The λ-calculus is a minimalistic language, where every value is a function, that can be used as a core calculus capturing the essential features of complex functional programming languages. It was formulated by Alonzo Church [Chu36] in the 1930s as a model of computability. At about the same time, Alan Turing was formulating what is now known as a Turing machine [Tur36] for the same purpose. It was later proved that both systems are equally expressive [Tur37].

As a programming language, the λ-calculus can be intriguing at first because everything reduces to function abstraction and application. The syntax comprises three sorts of terms: variables, function abstractions over a variable and applications of a term to an other. Those three constructs are summarized in the following grammar:

$t ::=$

x	variable
$\lambda x.\ t$	abstraction
$t_1\ t_2$	application

Below are a few standard λ-terms shown as examples of how the grammar is actually used:[3]

$\lambda x.\ x$	identity
$\lambda x.\ \lambda y.\ x$	constant
$\lambda f.\ \lambda x.\ f\ x\ x$	double application
$\lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x)$	function composition

The λ-calculus having no built-in constant or primitive operators (e.g. numbers, arithmetic operations, conditionals, loops, etc.), the only way to compute a value is by function application, also known as β-reduction and denoted by $\rightarrow_\beta$). It consists of replacing every instance of the abstracted variable in the abstraction body by the provided argument. Following is an

[3]To reduce the need for parentheses, we use the following standard conventions: (1) the body of a λ-abstraction expands as far as possible to the right and (2) function application is left-associative.

example of a λ-term that is β-reduced three times:

$$\begin{aligned}(\lambda x.\ (\lambda y.\ y)\ x)\ ((\lambda w.\ w)\ z) &\rightarrow_\beta (\lambda x.\ (\lambda y.\ y)\ x)\ z \\ &\rightarrow_\beta (\lambda y.\ y)\ z \\ &\rightarrow_\beta z\end{aligned}$$

This apparently simple operation hides a subtle corner case: name clashes. Nothing prevents two functions from using the same name for their abstracted variable. One cannot simply replace every variable with the same name when performing a substitution but must also take variable scopes into account. Here is a simple example that demonstrates that a naive approach can fail to preserve the semantics of the original λ-term:

$$(\lambda x.\ \lambda y.\ x)\ y \not\rightarrow_\beta \lambda y.\ y$$

One solution to this problem is to rename function arguments, also known as α-equivalence and denoted by $=_\alpha$, prior to β-reduction. Here is a correct β-reduction for the previous example:

$$\begin{aligned}(\lambda x.\ \lambda y.\ x)\ y &=_\alpha (\lambda x.\ \lambda w.\ x)\ y \\ &\rightarrow_\beta \lambda w.\ y\end{aligned}$$

The difference is important: under the naive approach, the λ-term was wrongly reduced to the identity function while the correct reduction lead to a function returning the constant y.

Many constructions from high level programming languages can be encoded using only those basic features. Unary functions are already supported and n-ary functions can be straightforwardly emulated by having a function return another function, as was done in the previous examples:

$$n\text{-ary function} \equiv \lambda x_1.\ \lambda x_2.\ \cdots\ \lambda x_n.\ t$$

Another common construction is a *let binding* which serves to attach an identifier to a complex expression. It can be emulated in the λ-calculus with a single function abstraction:

$$\text{let } x = y \text{ in } t \equiv (\lambda x.\ t)\ y$$

Although significantly less obvious, it is also possible to express Booleans only with functions. The encoding is based on the idea that any use of Booleans can be expressed with only three primitives: a constant representing a true value, a constant representing a false value and an operation to choose between two options:

$$\begin{aligned} \text{true} &\equiv \lambda t.\ \lambda f.\ t \\ \text{false} &\equiv \lambda t.\ \lambda f.\ f \\ \text{if } b \text{ then } t \text{ else } e &\equiv \lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f \end{aligned}$$

Using the rules already discussed, it is easy to show that the following reduction is valid:[4]

$$\begin{aligned} \text{if true then } x \text{ else } y \equiv\ & (\lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f)\ \mathit{true}\ x\ y \\ \rightarrow_\beta^*\ & \mathit{true}\ x\ y \\ \equiv\ & (\lambda t.\ \lambda f.\ t)\ x\ y \\ \rightarrow_\beta^*\ & x \end{aligned}$$

Other encodings exist for constructions such as numbers, list, datatypes, arbitrary recursion, etc. For a more comprehensive introduction to the subject, Hankin's monograph [Han04] is a good starting point.

2.2 Type Systems

Type systems are a syntactic method to prove the absence of certain erroneous behaviors. They differ from testing in that they are exhaustive and compositional. In this context, exhaustiveness means that each checked invariant is proved for the complete program instead of focusing on a single unit of code. Compositionality means that proofs for individual components can be used to discharge an obligation about the interaction of the components.

The kinds of errors detected depend on the specific type system considered: they can range from fairly simple to very complex. Examples of simple errors include typographical mistakes, usage of values of the wrong kind and usage of undefined operations:[5]

add : $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$ true : $\mathbb{B}$ add true true	Error in function application, "add" expects a $\mathbb{N}$ as first argument but a $\mathbb{B}$ was provided.

[4]Multiple consecutive β-reductions are denoted with $\rightarrow_\beta^*$.

[5]The syntax $T_1 \rightarrow T_2$ is the standard notation for the type of a function with domain T_1 and codomain T_2. The $\rightarrow$ operator being right-associative, the type $T_1 \rightarrow (T_2 \rightarrow T_3)$ can be written without parentheses: $T_1 \rightarrow T_2 \rightarrow T_3$.

With sufficiently powerful type systems, specific requirements can be provided as type annotations. Examples of such include that the output of a sorting function is a permutation of its input, that the argument of an indexing operation on a list is in a valid range and that two multiplied matrices have compatible dimensions. The more information one puts in the types, the more invalid programs the type checker can catch. The interactive theorem prover Coq is based on this idea: each transformation, including theorems, is internally a type-transformation. The drawback of such expressive type systems is that more work is required by the programmer to convince the type checking algorithm that the program fulfills its specification. An important design decision when defining a type system is to find a tradeoff between those conflicting goals.

Since they are often bundled with the compiler of a programming language and, thus, part of the normal programming cycle, type systems allow early detection of programming errors. Moreover, the diagnoses of type checkers can often pointed accurately the source of the error, unlike run-time tests where the effect of an error can sometime be visible only much further in the code when something starts to go wrong.

Another important way in which type systems can be used is as an abstraction tool. Large scale software generally consist of modules that communicate through interfaces. Types are a natural fit to serve as such an interface. Even in smaller scale programming, it is useful to characterize a datatype not by the way it is implemented but by the different operations that can be perform on it. This focus on operations led to the concept of abstract datatypes.

Types can also be used an invaluable maintenance tool. They serve as a checked documentation of programs and, being simpler than the computations they characterize, they can help to reason about such computations on a higher level. But they can also serve a very practical purpose, by checking which part of a program is affected by a change. If one decides to change the arguments of a function or to remove a field in a structure, a simple type checking pass will provide an exhaustive list of the places that must be updated.

Due to their static nature, type systems are normally conservative in that they will always reject bad programs at the expense of sometime rejecting good ones. A simple example of such limitation is the following program that fails to type-check, even though the Boolean expression will always evaluate to true at runtime:

if true then 42 else true	Type mismatch in conditional expression, the type of the “then” branch is $\mathbb{N}$ while the type of the “else” branch is $\mathbb{B}$.

Having only access to static information, a type checker only see that a Boolean can take two different values and, thus, must ensure that the program is valid in both cases. In this example, this implies that both branches of the “if” must be well typed and that their types must be compatible. It is the main goal of researchers on type systems to develop systems in which more valid programs are accepted while more invalid programs are rejected.

2.3 Isabelle/HOL

This section only presents briefly the most used constructions in the formalizations described in this thesis. It is not expected from the reader to fully understand the proofs presented in this report; it is sufficient to recognize key concepts such as induction and case analysis.

The theorem proving community is subdivided in two groups: automatic theorem proving (ATP) and interactive theorem proving (ITP). Each has its own set of goals, motivations, methodologies, tools and terminology. In ATP, one must formulate its context and equations in some logical formalism and ask the theorem prover to find a proof. The limiting factor is the algorithm used by the tool. Examples of such provers include SPASS, Vampire and Z3. In ITP, one must also formulate its context and equations in some logical formalism, although usually a more expressive one, but must also give instructions guiding the prover. The term proof assistant is sometime used to highlight this collaboration between the human and the machine. Here, the limiting factor is the ability of the human to guide its tool. Examples of interactive theorem provers include Agda, Coq and Isabelle. Isabelle is a generic interactive theorem prover for implementing logical formalisms and Isabelle/HOL is its specialization to a formalism called higher order logic.

An Isabelle theory file serves as the basic unit of encapsulation of formalizations and reusable libraries. It is analogous to modules in programming languages. Every definition and theorem developed must belong to a theory and can be made accessible to other theories by importing them.

Types and function definitions serve to describe entities and how to operate on them. They work in a very similar way to their counterpart in functional programming languages. A new type is introduced with the `datatype` command, followed by the name of the type and the different constructors separated by a `|`. Following is the standard definition of the type of parametric lists:[6]

```
datatype-new 'a list = Nil | Cons (head: 'a) (tail: 'a list)
```

The datatype consists of two constructors. The first one, *Nil*, is used to represent the empty list while the second one, *Cons*, is used to add an element in front of an existing list. Using those two primitives, a list can be created by successive application of the *Cons* constructor (e.g. *Cons 1* (*Cons 2* (*Cons 3 Nil*))). The *'a* in front of the name is a placeholder for a concrete type that must be provide later (e.g. the type of the previous example could be *nat list*).

Function definitions can take many forms in Isabelle/HOL. A primitively recursive function is introduced with the `primrec` command and defined by pattern matching over its arguments. Each pattern match entry must then provide the value to which the function evaluates. Following is a definition of a higher order function (i.e. a function that takes a function as argument) that checks if all the elements of a list are ordered with respect to a binary predicate provided as an argument:

```
primrec ordered :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ bool where
  ordered p Nil = True |
  ordered p (Cons x xs) = (case xs of
    Nil ⇒ True |
    Cons y ys ⇒ p x y ∧ ordered p xs)
```

An alternative way in which this function could be defined is inductively. Introduced with the `inductive` command, it allows to express Boolean functions by providing an inductive definition of when the function should evaluate to true, leaving all the other cases to false. The definition consists of base cases and possibly many inductive cases. Following is the same function defined inductively:[7]

```
inductive ordered' :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ bool where
  empty-list:
    ordered' p Nil |
  singleton-list:
    ordered' p (Cons x Nil) |
  arbitrary-list:
    p x1 x2 ⟹ ordered' p (Cons x2 xs) ⟹ ordered' p (Cons x1 (Cons x2 xs))
```

[6]Prefixing an element with a descriptive name, as done for the arguments of the *Cons* constructor, is a common pattern in Isabelle/HOL. The syntax is always `name : element`.

[7]Here, a descriptive name have been given to each rule. The base cases are *empty-list* and *singleton-list* while the inductive case is *arbitrary-list*.

Theorems, also named lemmas, are true facts involving the defined elements. They could be compared to the assertions used in programming languages. The main difference is that asserts are validate by evaluating the expression while theorems are validated by a formal proof. This is the point where the analogy with programming languages stops since this concept is unique to theorem provers. In Isabelle/HOL, a proof can take two forms: a low level sequence of `apply` steps or higher level structured definition. Following is an example of a lemma that proves, using the low level style, that the list of increasing natural numbers are ordered with respect to the usual comparison operation:

```
lemma ordered (λx (y :: nat). x < y) (Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil))))
  apply (unfold ordered.simps)
  apply (unfold list.case)
  apply (rule conjI, simp)+
  apply (rule TrueI)
done
```

This proof is easily checked by a computer but very hard to read for a human. For this reason, the alternative structured Isar proof language was designed to allow the writing of more human-friendly proofs. Following is a theorem showing that, whenever the *ordered* function returns true for a given predicate and list, the *ordered'* function will also return true:[8]

```
lemma primrec-imp-inductive:
  ordered f xs ⟹ ordered' f xs
proof (induction xs rule: list.induct)
  case Nil
  thus ?case by (auto intro: ordered'.intros)
next
  case (Cons y ys)
  thus ?case by (cases ys rule: list.exhaust) (auto intro: ordered'.intros)
qed
```

This proof is still easily checked by a computer but is also more readable for a human. It is easy to see that the proof works by induction on the list *xs*, that the base case (*Nil*) is first proved and that in the inductive case (*Cons*), a cases analysis of the values the argument *ys* can take is performed.

For a more comprehensive introduction to Isabelle/HOL, the reader is encourage to start with the first part of the book *Concrete Semantics* [NK14] and continue, for a deeper understanding, with the more exhaustive tutorial [NPW14] distributed with the system.

[8]In Isabelle, every unbound term is implicitly universally quantified: $n + 1 > n \equiv (\forall n.\ n + 1 > n)$.

3 Structure of the Formalization

In this thesis, we formalize six chapters of the first two sections of *TAPL*. Figure 1 presents the table of contents of those two sections — the formalized chapters are in bold — and Figure 2 presents the dependencies between the chapters; a normal arrow implies a direct dependency while a dashed arrow only imply that the knowledge learned in one chapter is reused.

I Untyped Systems

- § 3 **Untyped Arithmetic Expressions**
- § 4 An ML Implementation of Arithmetic Expressions
- § 5 **The Untyped Lambda-Calculus**
- § 6 **Nameless Representation of Terms**
- § 7 An ML Implementation of the Lambda-Calculus

II Simple Types

- § 8 **Typed Arithmetic Expressions**
- § 9 **Simply Typed Lambda-Calculus**
- § 10 An ML implementation of Simple Types
- § 11 Simple Extensions
- § 12 Normalization
- § 13 References
- § 14 Exceptions

Figure 1: Part I and II of *TAPL*

$$\begin{tikzcd}
 & & \boxed{3} & \\
 & \boxed{5} \arrow[ur, dashed] & & \boxed{8} \arrow[ul] \\
\boxed{6} \arrow[ur] & & \boxed{9} \arrow[ul] \arrow[ur, dashed] &
\end{tikzcd}$$

Figure 2: Dependencies between the chapters of TAPL

$$\begin{tikzcd}
\boxed{6} & & \boxed{3} & \\
 & \boxed{5} \arrow[ul] \arrow[ur, dashed] & & \boxed{8} \arrow[ul] \\
 & & \boxed{9} \arrow[ul, dotted] \arrow[ur, dashed] &
\end{tikzcd}$$

Figure 3: Dependencies between the theory files

The formalization closely follows the structure of the book. We provide one Isabelle theory file per chapter and mainly introduce them in the same order. The only exception is the chapter on "Nameless Representation of Terms". In the book, it is treated as a completely separate issue from the chapters on λ-calculus (i.e. as an encoding that can be useful for an implementation). They present it as a concrete alternative to the implicit α-conversion they assume in their proofs, which is nice for humans but not rigorous enough for a computer. Even though a formalization is different from an implementation it has some similar requirements. Since we chose this nameless representation for our formalization, we must diverge from the book and introduce this subject before the untyped λ-calculus. Figure 3 presents the dependencies between our Isabelle theory files.

It is possible to directly base the typed arithmetic expressions on untyped arithmetic expression by importing the theory and reusing its definition. This reuse is possible because, for this language, types are external to the representation of terms. The same argument applies to the untyped λ-calculus. By contrast, for the typed λ-calculus, we need to alter the representation of terms to add the typing annotation on abstraction variables, thus preventing the reuse of the untyped λ-calculus theory. This is represented as a dotted arrow.

4 Untyped Arithmetic Expressions

The language of untyped arithmetic expressions consists of Boolean expressions, containing the constants `true` and `false` and conditionals as primitives, and natural numbers, containing the constant `zero`, the successor and predecessor functions and an operation to test equality with zero as primitives. Following the book, we start with a subset containing only the Boolean expression and carry on with fully fledged arithmetic expressions.

4.1 Booleans

The syntax of this language is defined, in the book, in the following way:

t ::=

true	constant true
false	constant false
if t then t else t	conditional

Its counterpart, using Isabelle/HOL's syntax, is a recursive datatype: [9]

datatype *bterm* =
  BTrue |
  BFalse |
  BIf bterm bterm bterm

The semantics of the language is defined using the small-step operational semantics which consists of an evaluation relation that performs the smallest possible step towards the final value. Values are a subset of terms that are considered as the final output of a computation. For the Booleans, the only values are the constants *BTrue* and *BFalse*. To describe these, the book uses the following notation:

t ::=	
true	true value
false	false value

We translate this in Isabelle/HOL using an inductive predicate that returns true if its argument is a value:

inductive *is-value-B* :: *bterm* ⇒ *bool* **where**
  is-value-B BTrue |
  is-value-B BFalse

The evaluation relation is concerned with the way a conditional expression will be reduced. The book uses the standard mathematical notation for inference rules:

$$\text{if true then } t_2 \text{ else } t_3 \implies t_2 \tag{1}$$

$$\text{if false then } t_2 \text{ else } t_3 \implies t_3 \tag{2}$$

$$\frac{t_1 \implies t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \implies \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \tag{3}$$

The first rule states that the evaluation of a conditional with a true condition leads to the "then" branch, the second rule states that the evaluation of a conditional with a false condition leads to the "else" branch and the third rule states that, if the condition is not a Boolean constant, it must be itself

[9]To prevent name clashes with Isabelle's predefined types and constants of the same name, our types and type constructors are prefixed with `b`, which stand for *Booleans*. Functions use a suffix for the same purpose.

evaluated. These rules translate easily into another inductive predicate that returns true if the first argument can be reduced in one step to the second argument:

```
inductive eval1-B :: bterm ⇒ bterm ⇒ bool where
  eval1-BIf-BTrue:
    eval1-B (BIf BTrue t2 t3) t2 |
  eval1-BIf-BFalse:
    eval1-B (BIf BFalse t2 t3) t3 |
  eval1-BIf:
    eval1-B t1 t1' ⟹ eval1-B (BIf t1 t2 t3) (BIf t1' t2 t3)
```

With these basic definitions, we can turn to the first theorem: the determinacy of one-step evaluation. This theorem states that the evaluation relation is deterministic (i.e. there is only one way in which a given term can be evaluate). The focus of this paper being on the definitions and theorems, we can skim over the proof, just highlighting that it goes by induction over the evaluation relation and that it involves some case analyses:

```
theorem eval1-B-determinacy:
  eval1-B t t' ⟹ eval1-B t t'' ⟹ t' = t''
proof (induction t t' arbitrary: t'' rule: eval1-B.induct)
  case (eval1-BIf-BTrue t1 t2)
  thus ?case by (auto elim: eval1-B.cases)
next
  case (eval1-BIf-BFalse t1 t2)
  thus ?case by (auto elim: eval1-B.cases)
next
  case (eval1-BIf t1 t1' t2 t3)
  from eval1-BIf.prems eval1-BIf.hyps show ?case
    by (auto dest: eval1-BIf.IH elim: eval1-B.cases)
qed
```

A key concept is that of normal form, for which the book gives the following definition:

> A term t is in *normal form* if no evaluation rule applies to it — i.e., if there is no t' such that $t \to t'$.

Since this definition mainly introduces some standard terminology for a property of terms with respect to the single-step evaluation relation, we translate it using a simple definition:

```
definition is-normal-form-B :: bterm ⇒ bool where
  is-normal-form-B t ⟷ (∀ t'. ¬ eval1-B t t')
```

We continue by proving that every value is in normal form:

```
theorem value-imp-normal-form:
  is-value-B t ⟹ is-normal-form-B t
by (auto elim: is-value-B.cases eval1-B.cases simp: is-normal-form-B-def)
```

For this simple language, the converse is also true: every term in normal form is a value. Our proof follows the book and use contradiction, structural induction over t and case analysis over the possible values.

```
theorem normal-form-imp-value:
  is-normal-form-B t ⟹ is-value-B t
by (rule ccontr, induction t rule: bterm.induct)
  (auto
    intro: eval1-B.intros is-value-B.intros
    elim: is-value-B.cases
    simp: is-normal-form-B-def)
```

The one-step evaluation is a useful representation of the semantic of a language, but it does not represent what really interests us: the final value of an evaluation. To this end, the book defines a multi-step evaluation relation based on the single-step one:

> The *multi-step evaluation* relation $\rightarrow^*$ is the reflexive, transitive closure of one-step evaluation. That is, it is the smallest relation such that (1) if t $t \rightarrow t'$ then $t \rightarrow^* t'$, (2) $t \rightarrow^* t$ for all t, and (3) if $t \rightarrow^* t'$ and $t' \rightarrow^* t''$, then $t \rightarrow^* t''$.

A direct translation to Isabelle/HOL would lead to the following definition:

```
inductive eval-direct :: bterm ⇒ bterm ⇒ bool where
  e-once:
    eval1-B t t' ⟹ eval-direct t t' |
  e-self:
    eval-direct t t |
  e-transitive:
    eval-direct t t' ⟹ eval-direct t' t'' ⟹ eval-direct t t''
```

However, this definition is inconvenient for theorem proving because it requires us to consider three cases for each induction on a evaluation relation. Instead, we choose to define the multi-step evaluation relation using a shape similar to a list of one-step evaluations. The inductive definition consists of a base case, the reflexive application, and of an inductive case where one step of evaluation is performed:

```
inductive eval-B :: bterm ⇒ bterm ⇒ bool where
  eval-B-base:
    eval-B t t |
  eval-B-step:
    eval1-B t t' ⟹ eval-B t' t'' ⟹ eval-B t t''
```

We then prove that this definition is equivalent to the direct translation of the definition found in the book:

```
lemma eval-B-once:
  eval1-B t t' ⟹ eval-B t t'
by (simp add: eval-B.intros)

lemma eval-B-transitive:
  eval-B t t' ⟹ eval-B t' t'' ⟹ eval-B t t''
by (induction t t' rule: eval-B.induct) (auto intro: eval-B.intros)

lemma eval-direct-eq-eval-B:
  eval-direct = eval-B
proof ((rule ext)+, rule iffI)
  fix t t'
  assume eval-direct t t'
  thus eval-B t t'
   by (auto intro: eval-B.intros elim: eval-direct.induct eval-B-once eval-B-transitive)
next
  fix t t'
  assume eval-B t t'
  thus eval-direct t t'
    by (auto intro: e-self dest!: e-once elim: eval-B.induct e-transitive)
qed
```

The next theorem we consider is the uniqueness of normal form, which is a corollary of the determinacy of the single-step evaluation:

```
corollary uniqueness-of-normal-form:
  eval-B t u ⟹ is-normal-form-B u ⟹
  eval-B t u' ⟹ is-normal-form-B u' ⟹
  u = u'
by (induction t u rule: eval-B.induct)
  (metis eval-B.cases is-normal-form-B-def eval1-B-determinacy)+
```

The last theorem we consider is the termination of evaluation. To prove it, we need first to add a helper lemma, which was implicitly assumed in the book, about the size of terms after evaluation:

lemma *eval-once-size-B*:
  eval1-B t t' $\Longrightarrow$ *size-B t* $>$ *size-B t'*
by (*induction t t' rule*: *eval1-B.induct*) *simp-all*

theorem *termination-of-evaluation*:
  $\exists\, t'$. *eval-B t t'* $\wedge$ *is-normal-form-B t'*
by (*induction rule*: *measure-induct-rule*[*of size-B*])
  (*metis eval-B.intros eval-once-size-B is-normal-form-B-def*)

4.2 Arithmetic Expressions

We now turn to the fully fledged arithmetic expression language. The syntax is defined in the same way as for Booleans:[10]

datatype *nbterm* =
  NBTrue |
  NBFalse |
  NBIf nbterm nbterm nbterm |
  NBZero |
  NBSucc nbterm |
  NBPred nbterm |
  NBIs-zero nbterm

Values now consist either of Booleans or numeric values, for which a separate inductive definition is given. Here is the definition as found in the book:

v ::=		
	true	true value
	false	false value
	nv	numeric value
nv ::=		
	0	zero value
	succ nv	successor value

Our inductive definition is very similar, but contains explicit assumptions on the nature of `nv`. The book uses naming conventions which define letters such as `t` as always representing terms, letters such as `v` as always representing values and variants of `nv` as always representing numeric values. In our formalization, such implicit assumption is possible for `t` because Isabelle/HOL infers that *nberm* is the only type that could be place at this position. Since

[10]The prefix *nb* stands for *numeric and Booleans.*

values and numeric values do not have a proper type but characterize a subset of terms, we must add assumptions to declare the nature of these variables:

inductive *is-numeric-value-NB* :: *nbterm* ⇒ *bool* **where**
  is-numeric-value-NB NBZero |
  is-numeric-value-NB nv ⟹ *is-numeric-value-NB* (*NBSucc nv*)

inductive *is-value-NB* :: *nbterm* ⇒ *bool* **where**
  is-value-NB NBTrue |
  is-value-NB NBFalse |
  is-numeric-value-NB nv ⟹ *is-value-NB nv*

The single-step evaluation relation is a superset of the one defined for Booleans:

inductive *eval1-NB* :: *nbterm* ⇒ *nbterm* ⇒ *bool* **where**
  — Rules relating to the evaluation of Booleans
  eval1-NBIf-NBTrue:
    eval1-NB (*NBIf NBTrue t2 t3*) *t2* |
  eval1-NBIf-NBFalse:
    eval1-NB (*NBIf NBFalse t2 t3*) *t3* |
  eval1-NBIf:
    eval1-NB t1 t1′ ⟹ *eval1-NB* (*NBIf t1 t2 t3*) (*NBIf t1′ t2 t3*) |

  — Rules relating to the evaluation of natural numbers
  eval1-NBSucc:
    eval1-NB t t′ ⟹ *eval1-NB* (*NBSucc t*) (*NBSucc t′*) |
  eval1-NBPred-NBZero:
    eval1-NB (*NBPred NBZero*) *NBZero* |
  eval1-NBPred-NBSucc:
    is-numeric-value-NB nv ⟹ *eval1-NB* (*NBPred* (*NBSucc nv*)) *nv* |
  eval1-NBPred:
    eval1-NB t t′ ⟹ *eval1-NB* (*NBPred t*) (*NBPred t′*) |

  — Rules relating to the evaluation of the test for equality with zero
  eval1-NBIs-zero-NBZero:
    eval1-NB (*NBIs-zero NBZero*) *NBTrue* |
  eval1-NBIs-zero-NBSucc:
    is-numeric-value-NB nv ⟹ *eval1-NB* (*NBIs-zero* (*NBSucc nv*)) *NBFalse* |
  eval1-NBIs-zero:
    eval1-NB t t′ ⟹ *eval1-NB* (*NBIs-zero t*) (*NBIs-zero t′*)

The multi-step evaluation relation and the definition of normal form are perfectly analogous to these for Booleans:

```
inductive eval-NB :: nbterm ⇒ nbterm ⇒ bool where
  eval-NB-base:
    eval-NB t t |
  eval-NB-step:
    eval1-NB t t′ ⟹ eval-NB t′ t″ ⟹ eval-NB t t″

definition is-normal-form-NB :: nbterm ⇒ bool where
  is-normal-form-NB t ⟷ (∀ t′. ¬ eval1-NB t t′)
```

The reason is that all the actual work is performed by the single-step evaluation relation.

In the book, the section covering this fully fledged arithmetic expression language is mainly an explanation of the constructions not present in the Boolean expression language and does not contains any proper theorems. Nevertheless, we revisit the properties introduced for the language of Booleans and either prove that they are still theorems or disprove them.

The determinacy of the single-step evaluation still holds:

```
theorem eval1-NB-determinacy:
  eval1-NB t t′ ⟹ eval1-NB t t″ ⟹ t′ = t″
proof (induction t t′ arbitrary: t″ rule: eval1-NB.induct)
  case (eval1-NBIf t1 t1′ t2 t3)
  from eval1-NBIf.prems eval1-NBIf.hyps show ?case
    by (auto intro: eval1-NB.cases dest: eval1-NBIf.IH)
next
  case (eval1-NBSucc t1 t2)
  from eval1-NBSucc.prems eval1-NBSucc.IH show ?case
    by (auto elim: eval1-NB.cases)
next
  case (eval1-NBPred-NBSucc nv1)
  from eval1-NBPred-NBSucc.prems eval1-NBPred-NBSucc.hyps show ?case
    by (cases rule: eval1-NB.cases)
      (auto
        intro: is-numeric-value-NB.intros
        elim: not-eval-once-numeric-value[rotated])
next
  case (eval1-NBPred t1 t2)
  from eval1-NBPred.hyps eval1-NBPred.prems show ?case
    by (auto
      intro: eval1-NBPred.IH is-numeric-value-NB.intros
      elim: eval1-NB.cases
      dest: not-eval-once-numeric-value)
next
```

```
  case (eval1-NBIs-zero-NBSucc nv)
  thus ?case by (auto
    intro: eval1-NB.cases not-eval-once-numeric-value is-numeric-value-NB.intros)
next
  case (eval1-NBIs-zero t1 t2)
  from eval1-NBIs-zero.prems eval1-NBIs-zero.hyps show ?case
    by (cases rule: eval1-NB.cases) (auto
      elim: eval1-NB.cases
      intro: eval1-NBIs-zero.IH is-numeric-value-NB.intros
      elim: not-eval-once-numeric-value[rotated])
qed (auto elim: eval1-NB.cases)
```

Every value is in normal form:

```
theorem value-imp-normal-form-NB:
  is-value-NB t ⟹ is-normal-form-NB t
by (auto
  intro: not-eval-once-numeric-value
  elim: eval1-NB.cases is-value-NB.cases
  simp: is-normal-form-NB-def)
```

But, unlike for Boolean expressions, some terms that are in normal form are not values. An example of such term is *NBSucc NBTrue*.

```
theorem not-normal-form-imp-value-NB:
  ∃ t. is-normal-form-NB t ∧ ¬ is-value-NB t (is ∃ t. ?P t)
proof
  have a: is-normal-form-NB (NBSucc NBTrue)
    by (auto elim: eval1-NB.cases simp: is-normal-form-NB-def)
  have b: ¬ is-value-NB (NBSucc NBTrue)
    by (auto elim: is-numeric-value-NB.cases simp: is-value-NB.simps)
  from a b show ?P (NBSucc NBTrue) by simp
qed
```

The uniqueness of normal form still holds:

```
corollary uniqueness-of-normal-form-NB:
  eval-NB t u ⟹ eval-NB t u' ⟹ is-normal-form-NB u ⟹ is-normal-form-NB
u' ⟹ u = u'
proof (induction t u arbitrary: u' rule: eval-NB.induct)
  case (eval-NB-base t)
  thus ?case by (auto elim: eval-NB.cases simp: is-normal-form-NB-def)
next
  case (eval-NB-step t1 t2 t3)
 thus ?case by (metis eval-NB.cases is-normal-form-NB-def eval1-NB-determinacy)
qed
```

So does the termination of the evaluation function:

```
theorem eval-NB-always-terminate:
  ∃ t'. eval-NB t t' ∧ is-normal-form-NB t'
proof (induction rule: measure-induct-rule[of size-NB])
  case (less t)
  show ?case
    apply (cases is-normal-form-NB t)
    apply (auto intro: eval-NB-base)
    using eval-NB-step eval-once-size-NB is-normal-form-NB-def less.IH
    by blast
qed
```

5 Nameless Representation of Terms

In the background section on λ-calculus (Section 2.1), we presented the problem of name clashes that can arise when performing β-reduction. In its definitions and proofs, the book only works up to α-equivalence: assuming that the variables would be implicitly renamed if such a name clash occurred. In a separate chapter, a different representation of terms that avoids such problem is presented. It is described as one possible encoding that can be used when implementing an compiler for the λ-calculus.

Even though we are not building a compiler, our computer-verified formalization requires us to explicitly handle this problem. We chose to use this representation and, thus must also formalize this chapter.

The idea behind this representation, known as de Bruijn indices, is to make variables reference directly their corresponding binder, rather than referring to them by name. This is accomplished by using an index that count the number of enclosing λ-abstractions between a variable and its binder. Following is an example of de Bruijn indices representation for the function composition combinator:

$$\lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x) \equiv \lambda\ \lambda\ \lambda\ 2\ (1\ 0)$$

This representation releases us from having to consider the case of variable name clashes at the expense of being harder to read and having to maintain the correct indices when adding and removing λ-abstractions. Using this representation, we define the syntax of the untyped lambda calculus as follow:[11]

[11]The prefix *ul* stands for *untyped lambda-calculus*.

datatype *ulterm* =
  ULVar nat |
  ULAbs ulterm |
  ULApp ulterm ulterm

Using this syntax, the same example of the function composition combinator looks like this:

ULAbs (*ULAbs* (*ULAbs* (*ULApp* (*ULVar 2*) (*ULApp* (*ULVar 1*) (*ULVar 0*)))))

We define a shift function serving to increase or decrease, by a fix amount d, all indices larger than c in a term:

primrec *shift-UL* :: *int* ⇒ *nat* ⇒ *ulterm* ⇒ *ulterm* **where**
  shift-UL d c (*ULVar k*) = *ULVar* (*if k* < *c then k else nat* (*int k* + *d*)) |
  shift-UL d c (*ULAbs t*) = *ULAbs* (*shift-UL d* (*Suc c*) *t*) |
  shift-UL d c (*ULApp t1 t2*) = *ULApp* (*shift-UL d c t1*) (*shift-UL d c t2*)

In this definition, there is a possible information loss. The variables use a natural number as index but the function allows to shift both up and down, thus the use of an integer for the shift increment. When a variable is encountered, we first convert the index from natural number to integer, which is always safe, perform the integer addition, which correspond to a subtraction if d is negative, and convert the result back to natural numbers to serve as the new index. This last operation converts negative numbers to zero. We know that this loss of information is safe, since it makes no sense to speak of negative indices. Our *shift-UL* function thus has an implicit assumption that it should not be called with a negative number larger than the smallest free variable in the term. Following is an example of shifting up every free variable by 2:

lemma *shift-UL 2 0*
  (*ULAbs* (*ULAbs* (*ULApp* (*ULVar 1*) (*ULApp* (*ULVar 0*) (*ULVar 2*))))) =
   ULAbs (*ULAbs* (*ULApp* (*ULVar 1*) (*ULApp* (*ULVar 0*) (*ULVar 4*))))
  by *simp*

On a first reading, the previous example may seems broken: the variables *ULVar 0* and *ULVar 1* are not incremented. This is because the shift function operates on free variables, i.e. variables whose index refers to a non-existing λ-abstraction. Since the binding referred by *ULVar 1* is in the term, it is not a free variable: it is bound.

We now define a substitution function that replaces every free variable with index j by the term s:

primrec *subst-UL* :: *nat* ⇒ *ulterm* ⇒ *ulterm* ⇒ *ulterm* **where**
subst-UL j s (*ULVar k*) = (*if k* = *j then s else ULVar k*) |
subst-UL j s (*ULAbs t*) = *ULAbs* (*subst-UL* (*Suc j*) (*shift-UL 1 0 s*) *t*) |
subst-UL j s (*ULApp t1 t2*) = *ULApp* (*subst-UL j s t1*) (*subst-UL j s t2*)

Here is an example of substituting the variable 0 by the variable 1:

lemma *subst-UL 0* (*ULVar 1*)
(*ULApp* (*ULVar 0*) (*ULAbs* (*ULAbs* (*ULVar 2*)))) =
ULApp (*ULVar 1*) (*ULAbs* (*ULAbs* (*ULVar 3*)))
by *simp*

Note that the indices are relative to their position in the term. This is why *ULVar 2* is also substituted in the previous example: counting the number of enclosing λ-abstractions shows us that this variable is, indeed, the same as *ULVar 0* outside the λ-abstractions. Of course, we must maintain this invariant by incrementing variables in our substituting term accordingly.

6 Untyped Lambda-Calculus

The untyped lambda calculus is the first core calculus we formalize. It imports the theory on the nameless representation of terms (Section 5), which formalizes the representation used for the syntax of the language. We complete the definitions, by providing the semantics, and we prove the determinacy of evaluation, the relation between values and normal form, the uniqueness of normal form and the potentially non-terminating nature of evaluation.

6.1 Definitions

In the pure λ-calculus, only function abstractions are considered values:

inductive *is-value-UL* :: *ulterm* ⇒ *bool* **where**
is-value-UL (*ULAbs t*)

Variables are not part of this definition because they are a way to refer to a specific λ-abstraction. Since abstractions are themselves values, we do not need to consider their bound variables. The only ones we could consider as values are the free variables, i.e. variables referring to non-existing λ-abstractions. In the following examples, every occurrence of w is free:

$$w \qquad (\lambda x.\ x)\ w \qquad (\lambda x. \lambda y. \lambda z.\ w\ x\ y\ z)$$

There is no consensus on how the semantics should handle such situations. By excluding them from the set of values, the semantics described in the book defines that such terms are meaningless. This decision is consistent with many programming languages where the use of an undefined identifier leads to an error, either at compile-time or at run-time.

The single-step evaluation relation is defined, in the book, with the following inference rules where $[x \mapsto s]\ t$ is the replacement of variable x by s in t:

$$\frac{t_1 \implies t_1'}{t_1\ t_2 \implies t_1'\ t_2} \tag{1}$$

$$\frac{t_2 \implies t_2'}{v_1\ t_2 \implies v_1\ t_2'} \tag{2}$$

$$(\lambda x.\ t_{12})\ v_2 \implies [x \mapsto v_2]\ t_{12} \tag{3}$$

The first rule states that the left side of an application must be reduced first, the second rule states that the right side of an application must be reduced second and the third rule states that an application consists of replacing both the λ-abstraction and the argument by the λ-abstraction's body where the substitution has been performed. We translate these rules with the following inductive definition:

```
inductive eval1-UL :: ulterm ⇒ ulterm ⇒ bool where
  eval1-ULApp1:
    eval1-UL t1 t1' ⟹ eval1-UL (ULApp t1 t2) (ULApp t1' t2) |
  eval1-ULApp2:
    is-value-UL v1 ⟹ eval1-UL t2 t2' ⟹ eval1-UL (ULApp v1 t2)
      (ULApp v1 t2') |
  eval1-ULApp-ULAbs:
    is-value-UL v2 ⟹ eval1-UL (ULApp (ULAbs t12) v2)
      (shift-UL (−1) 0 (subst-UL 0 (shift-UL 1 0 v2) t12))
```

Apart from the explicit assumption on the nature of *v1*, the only difference is the substitution in the third rule. This is the reason that motivated us to formalize the nameless representation of terms in the first place. The book uses a high level definition of substitution where name clashes are not considered. We replace this higher level operation by our concrete substitution operation on de Bruijn indices. We begin by shifting up by on the concrete argument because, conceptually, it *enters* the function abstraction. We then perform the proper substitution of the function's variable, i.e. of index zero. Finally, we shift down every variable of the resulting body to account for the removed λ-abstraction.

The multi-step evaluation relation and the normal form definitions follow the usual pattern:

```
inductive eval-UL :: ulterm ⇒ ulterm ⇒ bool where
  eval-UL-base:
    eval-UL t t |
  eval-UL-step:
    eval1-UL t t′ ⟹ eval-UL t′ t″ ⟹ eval-UL t t″

definition is-normal-form-UL :: ulterm ⇒ bool where
  is-normal-form-UL t ⟷ (∀ t′. ¬ eval1-UL t t′)
```

6.2 Theorems

In the book, this chapter consists mainly of the presentation of the λ-calculus, of which we gave a short introduction in the background section (Section 2.1), and does not contains meaningful theorems. Nevertheless, we revisit the properties introduced with the arithmetic expressions language (Section 4) and either prove that they are still theorems or disprove them.

The determinacy of the single-step evaluation still holds:

```
theorem eval1-UL-determinacy:
  eval1-UL t t′ ⟹ eval1-UL t t″ ⟹ t′ = t″
proof (induction t t′ arbitrary: t″ rule: eval1-UL.induct)
  case (eval1-ULApp1 t1 t1′ t2)
  from eval1-ULApp1.hyps eval1-ULApp1.prems show ?case
    by (auto elim: eval1-UL.cases is-value-UL.cases intro: eval1-ULApp1.IH)
next
  case (eval1-ULApp2 t1 t2 t2′)
  from eval1-ULApp2.hyps eval1-ULApp2.prems show ?case
    by (auto elim: eval1-UL.cases is-value-UL.cases intro: eval1-ULApp2.IH)
next
  case (eval1-ULApp-ULAbs v2 t12)
  thus ?case by (auto elim: eval1-UL.cases simp: is-value-UL.simps)
qed
```

Every value is in normal form:

```
theorem value-imp-normal-form:
  is-value-UL t ⟹ is-normal-form-UL t
by (auto elim: is-value-UL.cases eval1-UL.cases simp: is-normal-form-UL-def)
```

Meanwhile, the converse of the preceding theorem is not true since variables are in normal form but are not values:

theorem *normal-form-does-not-imp-value*:
  $\exists t.$ *is-normal-form-UL* $t \wedge \neg$ *is-value-UL* t (**is** $\exists t.$ *?P* t)
proof
  have *a*: $\bigwedge n.$ *is-normal-form-UL* (*ULVar* n)
    by (*auto simp*: *is-normal-form-UL-def elim*: *eval1-UL.cases*)
  have *b*: $\bigwedge n. \neg$ *is-value-UL* (*ULVar* n)
    by (*auto simp*: *is-normal-form-UL-def elim*: *is-value-UL.cases*)
  from *a b* **show** $\bigwedge n.$ *?P* (*ULVar* n) **by** *simp*
qed

The uniqueness of normal form still holds:

corollary *uniqueness-of-normal-form*:
  eval-UL t $u \Longrightarrow$ *eval-UL* t $u' \Longrightarrow$ *is-normal-form-UL* $u \Longrightarrow$ *is-normal-form-UL* $u' \Longrightarrow u = u'$
by (*induction* t u *rule*: *eval-UL.induct*)
  (*metis eval-UL.cases is-normal-form-UL-def eval1-UL-determinacy*)+

This time, the evaluation relation could be non-terminating. A typical example of term whose evaluation does not terminate is the self-application combinator ($\omega \equiv \lambda x.\ x\ x$) applied to itself, resulting in a term called Ω:

definition ω :: *ulterm* **where**
  $\omega \equiv$ *ULAbs* (*ULApp* (*ULVar 0*) (*ULVar 0*))

definition Ω :: *ulterm* **where**
  $\Omega \equiv$ *ULApp* ω ω

A single step of evaluation will result in the same term:

lemma *eval1-UL-*Ω:
  eval1-UL Ω $t \Longrightarrow \Omega = t$
by (*induction* Ω t *rule*: *eval1-UL.induct*)
  (*auto elim*: *eval1-UL-ULAbsD simp*: ω*-def* Ω*-def*)

Since the single-step evaluation is equivalent to the identity, the multi-step evaluation relation will loop infinitely (e.g. $\Omega \rightarrow \Omega \rightarrow \ldots$):

lemma *eval-UL-*Ω:
  eval-UL Ω $t \Longrightarrow \Omega = t$
by (*induction* Ω t *rule*: *eval-UL.induct*) (*blast dest*: *eval1-UL-*Ω)+

lemma
  eval-UL Ω Ω
by (*rule eval-UL.intros*)

Based on this simple example, we can show that there exists some terms which cannot be reduce to a normal form:

theorem *eval-does-not-always-terminate*:
  $\exists\, t.\ \forall\, t'.$ *eval-UL* $t\ t' \longrightarrow \neg$ *is-normal-form-UL* t' (**is** $\exists\, t.\ \forall\, t'.$ *?P* $t\ t'$)
proof
  show $\forall\, t'.$ *?P* $\Omega\ t'$
    by (*auto dest*!: *eval-UL*-Ω)
      (*auto*
        intro: *eval1-UL.intros is-value-UL.intros*
        simp: ω-*def* Ω-*def is-normal-form-UL-def*)
qed

7 Typed Arithmetic Expressions

In this section, we revisit the previously formalized arithmetic expression language (Section 4) and augment it with static types. Since types are a characterization external to the definition of terms, we import the theory to reuse its definitions and theorems. We complete the definitions with the typing relation and prove type safety through the progress and preservation theorems.

7.1 Definitions

The language of arithmetic expressions contains two types for Booleans and natural numbers, which we model using a datatype:

datatype *nbtype* = *Bool* | *Nat*

The typing relation serves to assign a type to an expression. It is characterized by the following inference rules:

$$\text{true} : \text{Bool} \tag{1}$$

$$\text{false} : \text{Bool} \tag{2}$$

$$\frac{t_1 : \text{Bool} \qquad t_2 : \text{T} \qquad t_3 : \text{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \text{T}} \tag{3}$$

$$0 : \text{Nat} \tag{4}$$

$$\frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}} \tag{5}$$

$$\frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}} \tag{6}$$

$$\frac{t_1 : \text{Nat}}{\text{iszero } t_1 : \text{Bool}} \tag{7}$$

The first, second and fourth rules give the type of constants. The third rule requires that both branches of a conditional have the same type and that the condition is a Boolean. The fifth and sixth rules state that the successor and predecessor of natural numbers are natural numbers themselves. Finally, the seventh rule state that the test of equality with zero requires a natural number and leads a Boolean. We translate these rules in an inductive definition, for which we also provide the |:| operator as a more conventional notation:

```
inductive has-type :: nbterm ⇒ nbtype ⇒ bool (infix |:| 150) where
  — Rules relating to the type of Booleans
  has-type-NBTrue:
    NBTrue |:| Bool |
  has-type-NBFalse:
    NBFalse |:| Bool |
  has-type-NBIf:
    t1 |:| Bool ⟹ t2 |:| T ⟹ t3 |:| T ⟹ NBIf t1 t2 t3 |:| T |

  — Rules relating to the type of natural numbers
  has-type-NBZero:
    NBZero |:| Nat |
  has-type-NBSucc:
    t |:| Nat ⟹ NBSucc t |:| Nat |
  has-type-NBPred:
    t |:| Nat ⟹ NBPred t |:| Nat |
  has-type-NBIs-zero:
    t |:| Nat ⟹ NBIs-zero t |:| Bool
```

The inversion of the typing relation gives us information on types for specific terms:

```
lemma inversion-of-typing-relation:
  NBTrue |:| R ⟹ R = Bool
  NBFalse |:| R ⟹ R = Bool
  NBIf t1 t2 t3 |:| R ⟹ t1 |:| Bool ∧ t2 |:| R ∧ t3 |:| R
  NBZero |:| R ⟹ R = Nat
```

  NBSucc t |:| *R* $\Longrightarrow$ *R* = *Nat* $\wedge$ *t* |:| *Nat*
  NBPred t |:| *R* $\Longrightarrow$ *R* = *Nat* $\wedge$ *t* |:| *Nat*
  NBIs-zero t |:| *R* $\Longrightarrow$ *R* = *Bool* $\wedge$ *t* |:| *Nat*
by (*auto elim*: *has-type.cases*)

In the typed arithmetic language, every term t has at most one type. That is, if t is typable, then its type is unique:

theorem *uniqueness-of-types*:
  t |:| *T* $\Longrightarrow$ *t* |:| *T'* $\Longrightarrow$ *T* = *T'*
by (*induction t T rule*: *has-type.induct*) (*auto dest*: *inversion-of-typing-relation*)

7.2 Safety = Progress + Preservation

The most basic property a type system must provide is *safety*, also called *soundness*: the evaluation of a well-typed term will not reach a state whose semantics is undefined. Since our *operational semantics* is based the of the evaluation relation and the value predicate, every term that does not fit in one or the other has no defined semantics.

An example of an undefined state is *NBSucc NBTrue*: there is no further evaluation step possible but it is not a value neither. In our current language, there is nothing we can do with this term.

Another usefull lemma is the canonical form of values which, for well typed terms, give us information on the nature of the terms:

lemma *canonical-form*:
  is-value-NB v $\Longrightarrow$ *v* |:| *Bool* $\Longrightarrow$ *v* = *NBTrue* $\vee$ *v* = *NBFalse*
  is-value-NB v $\Longrightarrow$ *v* |:| *Nat* $\Longrightarrow$ *is-numeric-value-NB v*
by (*auto elim*: *has-type.cases is-value-NB.cases is-numeric-value-NB.cases*)

The safety of a type system can be shown in two step: progress and preservation. Progress means that a well-typed term is not stuck, i.e. either it is a value or it can take a step according to the evaluation rules.

theorem *progress*:
  t |:| *T* $\Longrightarrow$ *is-value-NB t* $\vee$ ($\exists$ *t'*. *eval1-NB t t'*)
proof (*induction t T rule*: *has-type.induct*)
  case (*has-type-NBPred t*)
  thus *?case*
   by (*auto intro*: *eval1-NB.intros is-numeric-value-NB.cases dest*: *canonical-form*)
next
  case (*has-type-NBIs-zero t*)
  thus *?case*
   by (*auto intro*: *eval1-NB.intros is-numeric-value-NB.cases dest*: *canonical-form*)

qed (*auto*
  intro: *eval1-NB.intros is-value-NB.intros is-numeric-value-NB.intros*
  dest: *canonical-form*)

Preservation means that if a well-typed term takes a step of evaluation, then the resulting term is also well-typed.

theorem *preservation*: t |:| $T \Longrightarrow$ *eval1-NB* $t\ t' \Longrightarrow t'$ |:| T
proof (*induction t T arbitrary*: t' *rule*: *has-type.induct*)
  case (*has-type-NBIf t1 t2 T t3*)
  from *has-type-NBIf.prems has-type-NBIf.IH has-type-NBIf.hyps* **show** *?case*
    by (*auto intro*: *has-type.intros elim*: *eval1-NB.cases*)
qed (*auto*
  intro: *has-type.intros*
  dest: *inversion-of-typing-relation*
  elim: *eval1-NB.cases*)

8 Typed Lambda Calculus

We now revisit the λ-calculus (Section 6) and augment it with static types. Unlike the typed arithmetic expressions language, types are an integral part of the language and its syntax. For this reason, we cannot import the theory of the untyped variant and build on top of it, but need to provide new, although similar, definitions. We will prove type safety through the progress and preservation theorems before showing that types can be safely erased while preserving the semantics of the language.

8.1 Definitions

In the untyped lambda-calculus, everything is a function. Thus, we need to provide the type of functions, usually written $a \to b$ which, given an argument of type a, will evaluate to a value of type b. Since both a and b must be valid types, we need to provide a base case to stop the recursion at some point. To keep the language minimal, we only add the Boolean type as a base case:[12]

datatype-new *ltype* =
  Bool |
  Fun (*domain*: *ltype*) (*codomain*: *ltype*) (**infixr** $\to$ *225*)

[12]The prefix *l* stands for *lambda-calculus.*

In the previous definition, → is a type constructor which can be use to create function types for some concrete domain and codomain. Examples of such types include the following:[13]

$$Bool \to Bool \tag{1}$$
$$(Bool \to Bool) \to Bool \tag{2}$$
$$(Bool \to Bool) \to (Bool \to Bool) \tag{3}$$
$$(Bool \to Bool) \to Bool \to Bool \tag{4}$$

Programming languages usually have more than our base type. Examples include integers, floating point numbers, characters, arrays, etc.

Since variables can now range over infinitely many types, we need a way to know which type a function requires as domain. There are two possible strategies: we can annotate the λ-abstractions with the intended type of their arguments, or else we can analyze the body of the abstraction to infer the required type. *TAPL* chose the former strategy.

The syntax of this language differs from the pure λ-calculus by having constructions for Boolean expressions and a type annotation on function abstractions:

```
datatype-new lterm =
  LTrue |
  LFalse |
  LIf (bool-expr: lterm) (then-expr: lterm) (else-expr: lterm) |
  LVar nat |
  LAbs (arg-type: ltype) (body: lterm) |
  LApp lterm lterm
```

We define the shift and substitution functions for this extended language:

```
primrec shift-L :: int ⇒ nat ⇒ lterm ⇒ lterm where
  shift-L d c LTrue = LTrue |
  shift-L d c LFalse = LFalse |
  shift-L d c (LIf t1 t2 t3) = LIf (shift-L d c t1) (shift-L d c t2) (shift-L d c t3) |
  shift-L d c (LVar k) = LVar (if k < c then k else nat (int k + d)) |
  shift-L d c (LAbs T t) = LAbs T (shift-L d (Suc c) t) |
  shift-L d c (LApp t1 t2) = LApp (shift-L d c t1) (shift-L d c t2)

primrec subst-L :: nat ⇒ lterm ⇒ lterm ⇒ lterm where
  subst-L j s LTrue = LTrue |
  subst-L j s LFalse = LFalse |
```

[13]Note that the last two examples are equivalent, since the → operator is right-associative.

subst-L j s $($*LIf t1 t2 t3*$)$ $=$ *LIf* $($*subst-L j s t1*$)$ $($*subst-L j s t2*$)$ $($*subst-L j s t3*$)$ |
subst-L j s $($*LVar k*$)$ $=$ $($*if k* $=$ *j then s else LVar k*$)$ |
subst-L j s $($*LAbs T t*$)$ $=$ *LAbs T* $($*subst-L* $($*Suc j*$)$ $($*shift-L 1 0 s*$)$ *t*$)$ |
subst-L j s $($*LApp t1 t2*$)$ $=$ *LApp* $($*subst-L j s t1*$)$ $($*subst-L j s t2*$)$

The semantics is similar to the pure λ-calculus. A first difference is that the set of values also contain the Boolean constants:

inductive *is-value-L* :: *lterm* $\Rightarrow$ *bool* **where**
is-value-L LTrue |
is-value-L LFalse |
is-value-L $($*LAbs T t*$)$

A second difference is that the single-step evaluation relation also contains the rules for the evaluation of the conditional statement:

inductive *eval1-L* :: *lterm* $\Rightarrow$ *lterm* $\Rightarrow$ *bool* **where**
— Rules relating to the evaluation of Booleans
eval1-LIf-LTrue:
eval1-L $($*LIf LTrue t2 t3*$)$ *t2* |
eval1-LIf-LFalse:
eval1-L $($*LIf LFalse t2 t3*$)$ *t3* |
eval1-LIf:
eval1-L t1 t1' $\Longrightarrow$ *eval1-L* $($*LIf t1 t2 t3*$)$ $($*LIf t1' t2 t3*$)$ |

— Rules relating to the evaluation of function application
eval1-LApp1:
eval1-L t1 t1' $\Longrightarrow$ *eval1-L* $($*LApp t1 t2*$)$ $($*LApp t1' t2*$)$ |
eval1-LApp2:
is-value-L v1 $\Longrightarrow$ *eval1-L t2 t2'* $\Longrightarrow$ *eval1-L* $($*LApp v1 t2*$)$ $($*LApp v1 t2'*$)$ |
eval1-LApp-LAbs:
is-value-L v2 $\Longrightarrow$ *eval1-L* $($*LApp* $($*LAbs T t12*$)$ *v2*$)$
$($*shift-L* (-1) *0* $($*subst-L 0* $($*shift-L 1 0 v2*$)$ *t12*$))$

When type checking the body of a function abstraction, we assume that the given function argument does have the type annotated. Since the body could itself be a function abstraction, we need to keep track of this set of typing assumptions, also known as a typing context. Since the book considers variables to be a named reference to a λ-abstraction, its typing context is a set of identifier–type pairs. Our use of de Bruijn indices requires us to consider an alternative representation. We define a context to be a list of types whose nth position contains the type of the nth free variale:

type-synonym *lcontext* $=$ *ltype list*

To keep the notation similar to the book, we define some synonyms for the list operations that mimic their set counterpart:

notation *Nil* (∅)
abbreviation *cons* :: *lcontext* ⇒ *ltype* ⇒ *lcontext* (**infixl** |,| *200*) **where**
  cons Γ *T* ≡ *T* # Γ
abbreviation *elem′* :: (*nat* × *ltype*) ⇒ *lcontext* ⇒ *bool* (**infix** |∈| *200*) **where**
  elem′ p Γ ≡ *fst p* < *length* Γ ∧ *snd p* = *nth* Γ (*fst p*)

With the concept of typing concept, the syntax used for the typing relation needs to be extended:

$$\Gamma \vdash t : T$$

This syntax can be read as "under the context Γ, the term *t* have type *T*. We now define the typing relation by translating the induction rules present in the book to an inductive definition:

inductive *has-type-L* :: *lcontext* ⇒ *lterm* ⇒ *ltype* ⇒ *bool* (((-)/ ⊢ (-)/ |:| (-)) [*150*, *150*, *150*] *150*) **where**
  — Rules relating to the type of Booleans
  has-type-LTrue:
    Γ ⊢ *LTrue* |:| *Bool* |
  has-type-LFalse:
    Γ ⊢ *LFalse* |:| *Bool* |
  has-type-LIf:
    Γ ⊢ *t1* |:| *Bool* ⟹ Γ ⊢ *t2* |:| *T* ⟹ Γ ⊢ *t3* |:| *T* ⟹ Γ ⊢ (*LIf t1 t2 t3*) |:| *T* |

  — Rules relating to the type of the constructs of the λ-calculus
  has-type-LVar:
    (*x*, *T*) |∈| Γ ⟹ Γ ⊢ (*LVar x*) |:| *T* |
  has-type-LAbs:
    (Γ |,| *T1*) ⊢ *t2* |:| *T2* ⟹ Γ ⊢ (*LAbs T1 t2*) |:| (*T1* → *T2*) |
  has-type-LApp:
    Γ ⊢ *t1* |:| (*T11* → *T12*) ⟹ Γ ⊢ *t2* |:| *T11* ⟹ Γ ⊢ (*LApp t1 t2*) |:| *T12*

The rules for Booleans are the same as in section 7. The rule *has-type-LVar* states that the type of a variable must be in the typing context. The rule *has-type-LAbs* states that the type of an λ-abstraction depends on the type of both its argument and body. Finally, the rule *has-type-LApp* states that the type of a function application is the codomain of the function. As an example of a usage of the typing relation, consider the type of the application of *LTrue* to the Boolean identity function:

lemma ∅ ⊢ (*LApp* (*LAbs Bool* (*LVar 0*)) *LTrue*) |:| *Bool*
  by (*auto intro*!: *has-type-L.intros*)

A more interesting example, assuming there is one variable of type *Bool* → *Bool* in the typing context, is the type of applying a Boolean expression to this variable:

lemma
  assumes $\Gamma = \emptyset$ |,| (*Bool* → *Bool*)
  shows $\Gamma \vdash$ *LApp* (*LVar 0*) (*LIf LFalse LTrue LFalse*) |:| *Bool*
by (*auto intro*!: *has-type-L.intros simp*: *assms*)

8.2 Properties of Typing

The inversion of typing relation, which gives us information on types for specific terms, will be a useful lemma in the following theorems:

lemma *inversion*:
  $\Gamma \vdash$ *LTrue* |:| *R* ⟹ *R* = *Bool*
  $\Gamma \vdash$ *LFalse* |:| *R* ⟹ *R* = *Bool*
  $\Gamma \vdash$ *LIf t1 t2 t3* |:| *R* ⟹ $\Gamma \vdash$ *t1* |:| *Bool* ∧ $\Gamma \vdash$ *t2* |:| *R* ∧ $\Gamma \vdash$ *t3* |:| *R*
  $\Gamma \vdash$ *LVar x* |:| *R* ⟹ (*x*, *R*) |∈| Γ
  $\Gamma \vdash$ *LAbs T1 t2* |:| *R* ⟹ ∃ *R2*. *R* = *T1* → *R2* ∧ Γ |,| *T1* ⊢ *t2* |:| *R2*
  $\Gamma \vdash$ *LApp t1 t2* |:| *R* ⟹ ∃ *T11*. Γ ⊢ *t1* |:| *T11* → *R* ∧ Γ ⊢ *t2* |:| *T11*
  by (*auto elim*: *has-type-L.cases*)

Every term has at most one type:

theorem *uniqueness-of-types*:
  Γ ⊢ *t* |:| *T1* ⟹ Γ ⊢ *t* |:| *T2* ⟹ *T1* = *T2*
by (*induction* Γ *t T1 arbitrary*: *T2 rule*: *has-type-L.induct*)
  (*metis prod.sel ltype.sel inversion*)+

The canonical form of values, which gives us information on terms for well-typed values, will also be useful later:

lemma *canonical-forms*:
  is-value-L v ⟹ Γ ⊢ *v* |:| *Bool* ⟹ *v* = *LTrue* ∨ *v* = *LFalse*
  is-value-L v ⟹ Γ ⊢ *v* |:| *T1* → *T2* ⟹ ∃ *t*. *v* = *LAbs T1 t*
by (*auto elim*: *has-type-L.cases is-value-L.cases*)

To formalize the concept of free variables (i.e. variables referring to a non existing λ-abstraction), we provide a function that return the set of free variables of a term:

primrec *FV* :: *lterm* ⇒ *nat set* **where**
  FV LTrue = {} |
  FV LFalse = {} |
  FV (*LIf t1 t2 t3*) = *FV t1* ∪ *FV t2* ∪ *FV t3* |
  FV (*LVar x*) = {*x*} |

FV (*LAbs T t*) = *image* ($\lambda x.\ x - 1$) (*FV t* − {*0*}) |
FV (*LApp t1 t2*) = *FV t1* ∪ *FV t2*

Based on the FV function, we can now define a closed term to be a term whose set of free-variables is empty:

definition *is-closed* :: *lterm* ⇒ *bool* **where**
is-closed t ≡ *FV t* = {}

We now prove the progress theorem (i.e. a well-typed closed term is either a value or can take a step according to the evaluation rules):

theorem *progress*:
∅ ⊢ *t* |:| *T* ⟹ *is-closed t* ⟹ *is-value-L t* ∨ (∃ *t'*. *eval1-L t t'*)
proof (*induction t T rule*: *has-type-L.induct*)
case (*has-type-LIf* Γ *t1 t2 T t3*)
thus *?case* **by** (*cases is-value-L t1*)
(*auto intro*: *eval1-L.intros dest*: *canonical-forms simp*: *is-closed-def*)
next
case (*has-type-LApp* Γ *t1 T11 T12 t2*)
thus *?case* **by** (*cases is-value-L t1*, *cases is-value-L t2*)
(*auto intro*: *eval1-L.intros dest*: *canonical-forms simp*: *is-closed-def*)
qed (*simp-all add*: *is-value-L.intros is-closed-def*)

Proving the preservation theorem requires us to first prove a number of helper lemmas. For these, our reliance on "de Bruijn indices" forces us to depart substantially from the book.

The first lemma the book considers is the permutation of the typing context:

> If $\Gamma \vdash t : T$ and Δ is a permutation of Γ, then $\Delta \vdash t : T$. Moreover, the latter derivation has the same depth as the former.

Translated naively, this lemma does not hold with our representation of the typing context as an ordered. Instead, we will prove an other lemma which states that it is safe to remove a variable from the context if it is not referenced in the considered term:

lemma *shift-down*:
insert-nth n U Γ ⊢ *t* |:| *T* ⟹ *n* ≤ *length* Γ ⟹
(⋀*x*. *x* ∈ *FV t* ⟹ *x* ≠ *n*) ⟹ Γ ⊢ *shift-L* (− *1*) *n t* |:| *T*
proof (*induction insert-nth n U* Γ *t T arbitrary*: Γ *n rule*: *has-type-L.induct*)
case (*has-type-LAbs V t T*)
from *this*(*1*,*3*,*4*) **show** *?case*
by (*fastforce intro*: *has-type-L.intros has-type-LAbs.hyps*(*2*)[**where** *n*=*Suc n*])+
qed (*fastforce intro*: *has-type-L.intros simp*: *nth-append min-def*)+

This lemma was the most challenging to express and prove. It was difficult to define the correct set of assumptions and, prior to simplifications, the proof was quite imposing.

The book then consider the weakening the typing context:

> If $\Gamma \vdash t : T$ and $x \notin dom(\Gamma)$, then $\Gamma, x : S \vdash t : T$. Moreover, the latter derivation has the same depth as the former.

This lemma does hold with our representation of the typing context, but we need to express it in terms of list by inserting the type S at a fixed position n. Moreover, we need to shift up every variable referring to a λ-abstraction further in the context than n.

```
lemma weakening:
  Γ ⊢ t |:| T ⟹ n ≤ length Γ ⟹ insert-nth n S Γ ⊢ shift-L 1 n t |:| T
proof (induction Γ t T arbitrary: n rule: has-type-L.induct)
  case (has-type-LAbs Γ T1 t2 T2)
  from has-type-LAbs.prems has-type-LAbs.hyps
    has-type-LAbs.IH[where n=Suc n] show ?case
    by (auto intro: has-type-L.intros)
qed (auto simp: nth-append min-def intro: has-type-L.intros)
```

This specific formulation was difficult to come with but the proof is, after simplifications, fairly short. It is a typical situation in interactive theorem proving that the result seems simple and does not make justice to the effort. It can be considered an achievement to reduce a huge and unreadable proof to a small and readable one.

The book then considers, as its last helper lemma, the preservation of types under substitution:

> If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s] : T$.

We prove a slightly different theorem that is more suitable for the coming proofs:

```
lemma substitution:
  Γ ⊢ t |:| T ⟹ Γ ⊢ LVar n |:| S ⟹ Γ ⊢ s |:| S ⟹ Γ ⊢ subst-L n s t |:| T
proof (induction Γ t T arbitrary: s n rule: has-type-L.induct)
  case (has-type-LAbs Γ T1 t T2)
  thus ?case by (fastforce
    intro: has-type-L.intros weakening[where n=0, unfolded insert-nth-def nat.rec]
    dest: inversion(4))
qed (auto intro!: has-type-L.intros dest: inversion(4))
```

We must provide a some more lemmas to define how the *FV* function behaves with respect to the *shift-L* and *subst-L* functions:

```
lemma FV-shift:
  FV (shift-L (int d) c t) = image (λx. if x ≥ c then x + d else x) (FV t)
proof (induction t arbitrary: c rule: lterm.induct)
  case (LAbs T t)
  thus ?case by (auto simp: gr-Suc-conv image-iff) force+
qed auto

lemma FV-subst:
  FV (subst-L n t u) = (if n ∈ FV u then (FV u − {n}) ∪ FV t else FV u)
proof (induction u arbitrary: n t rule: lterm.induct)
  case (LAbs T u)
  thus ?case
    apply (auto simp: gr0-conv-Suc image-iff FV-shift[of 1, unfolded int-1])
   by (metis DiffI One-nat-def UnCI diff-Suc-1 empty-iff imageI insert-iff nat.distinct(1))+
qed (auto simp: gr0-conv-Suc image-iff FV-shift[of 1, unfolded int-1])
```

Again, these lemmas are not present in the book. It is usual for paper proofs to be a little sketchy and rely on readers to imagine fill in the blanks with some simple lemmas. The need for these arise from the use of the *FV* function in the *shift-down* lemma.

Finally, we can now prove the preservation theorem:

```
theorem preservation:
  Γ ⊢ t |:| T ⟹ eval1-L t t′ ⟹ Γ ⊢ t′ |:| T
proof (induction Γ t T arbitrary: t′ rule: has-type-L.induct)
  case (has-type-LIf Γ t1 t2 T t3)
 thus ?case by (auto intro: has-type-L.intros eval1-L.cases[OF has-type-LIf.prems])
next
  case (has-type-LApp Γ t1 T11 T12 t2)
  thus ?case by (auto
    intro!: has-type-L.intros substitution shift-down
    dest!: inversion
    dest: weakening[where n=0, unfolded insert-nth-def nat.rec]
    elim!: eval1-LAppE
    split: lterm.splits if-splits
    simp: FV-subst FV-shift[of 1, unfolded int-1])
      (metis neq0-conv)
qed (auto elim: eval1-L.cases)
```

By proving the progress and the preservation theorems, we have shown that the typed λ-calculus is type safe, i.e. every well-typed program has a well-defined semantics.

8.3 Erasure and Typability

The type system we formalized is completely static (i.e. there is no run-time checked involving the types of terms). Since the type annotations are not used during evaluation, it is worth exploring the possibility to erase them prior to execution. To this end, we define an untyped version of our λ-calculus with Booleans:[14]

datatype *uterm* =
  UTrue |
  UFalse |
  UIf uterm uterm uterm |
  UVar nat |
  UAbs uterm |
  UApp uterm uterm

primrec *shift-U* :: *int* ⇒ *nat* ⇒ *uterm* ⇒ *uterm*
primrec *subst-U* :: *nat* ⇒ *uterm* ⇒ *uterm* ⇒ *uterm*
inductive *is-value-U* :: *uterm* ⇒ *bool*
inductive *eval1-U* :: *uterm* ⇒ *uterm* ⇒ *bool*

We now define a morphism which maps every typed term to an equivalent untyped one:

primrec *erase* :: *lterm* ⇒ *uterm* **where**
  erase LTrue = *UTrue* |
  erase LFalse = *UFalse* |
  erase (*LIf t1 t2 t3*) = (*UIf* (*erase t1*) (*erase t2*) (*erase t3*)) |
  erase (*LVar x*) = *UVar x* |
  erase (*LAbs - t*) = *UAbs* (*erase t*) |
  erase (*LApp t1 t2*) = *UApp* (*erase t1*) (*erase t2*)

We also characterize how the *erase* function reacts with respect to values and the *shift-L* and *subst-L* functions.

lemma *is-value-erasure*:
  is-value-L t = *is-value-U* (*erase t*)
by (*induction t rule*: *lterm.induct*) (*auto simp*: *is-value-L.simps is-value-U.simps*)

lemma *shift-erasure*:
  erase (*shift-L d c t*) = *shift-U d c* (*erase t*)
by (*induction t arbitrary*: *d c rule*: *lterm.induct*) *auto*

[14]The definitions are analogous to their typed counterpart.

lemma *subst-erasure*:
  erase (*subst-L j s t*) = *subst-U j* (*erase s*) (*erase t*)
by (*induction t arbitrary*: *j s rule*: *lterm.induct*) (*auto simp*: *shift-erasure*)

We can now prove that every evaluation step on a typed term can be performed in parallel on a corresponding untyped term.

theorem
  eval1-L t t' $\Longrightarrow$ *eval1-U* (*erase t*) (*erase t'*)
by (*induction t t' rule*: *eval1-L.induct*)
  (*auto intro*: *eval1-U.intros simp*: *shift-erasure subst-erasure is-value-erasure*)

9 Conclusion

In this thesis, we formalized a number of languages presented in *Types and Programming Languages* using the Isabelle/HOL interactive theorem prover. We started with a simple arithmetic language of Booleans and natural numbers. We continued with the nameless representation of terms for the λ-calculus, which we used as a basis for the pure untyped λ-calculus. For those languages, we proved the determinacy of evaluation, the relation between values and normal form, the uniqueness of normal form and the termination, or non-termination, of evaluation.

We then revisited both languages and augmented them with type systems. We proved the uniqueness of types and the safety of the languages through the progress and preservation theorems. We also demonstrated that the addition of types did not changed the semantics of the λ-calculus by proving that types can be erased without affecting the evaluation of terms.

A formalization can be separated in three main elements: definitions, properties involving these definitions and computer-checked proofs that these properties hold. In retrospective, expressing the definitions and properties was the most important and difficult activity. Once the right abstractions and the correct formulations for theorems were found, the proofs were usually fairly simple: a good definition is worth three theorems! Conversely, a wrong abstraction or hypothesis have led us to theorems very difficult to prove, or even to properties that ended up not being theorems at all.

In this report, we focused our attention on the definitions and theorems, highlighting the differences with the book. The complete Isabelle/HOL theories provided along with this report[15] contain more examples, exercises and less important theorems.

10 References

[Bla14] Jasmin C. Blanchette. *Hammering Away : A User's Guide to Sledgehammer for Isabele/HOL*, 2014.

[Bru72] Nicolaas G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[Han04] Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists*. College Publications, February 2004.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics*. Springer, 2014.

[NPW14] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL : A Proof Assistant for Higher-Order Logic*, 2014.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[Tur37] Alan M. Turing. Computability and lambda-definability. *The Journal of Symbolic Logic*, 2(4):153–163, 1937.

[15] https://github.com/authchir/log792-type-systems-formalization