Barendregt's λ-Cube in Isabelle

Marco Benini
Computational Architectures Laboratory
Computer Science Department – University of Milan

June 6, 1995

Abstract

We present an implementation of Barendregt's λ-Cube in the Isabelle proof system. Isabelle provides many facilities for developing a useful specification and proving environment from the basic formulation of formal systems. We used those facilities to provide an environment where the user can describe problems and derive proofs interactively.

1 Introduction

This paper describes a prover for typed λ-calculus. In particular we focus our attention to the eight variants in the Barendregt's system. There already exist other theorem provers for typed λ-calculus, but they don't pursue our goals. The principal aspect of most of them is efficiency of proof search. Instead, what we want to develop was an user friendly framework, where one could experiment with λ-calculus.

We have chosen to describe Barendregt's λ-cube because it contains the most used typed λ-calculi, and they are formalized in such a way that syntax of terms and most of the inference rules are identical among calculi.

We developed an extension of the basic syntax of Isabelle to mimic the form of λ-terms. We complemented the syntax with the appropriate inference rules, and we wrote some simple tactics to automate elementary proofs in those calculi.

We note that a partial work in the direction we are following, was made by Tobias Nipkow. In the standard distribution of Isabelle, a theory, *Cube*, is provided that implements the Barendregt's system. But we go further, since his formalization was based on an older work of Barendregt, and he didn't develop specific tactics for λ-calculus, nor documentation for the theory.

This paper can be regarded as a case study in the use of Isabelle as a logical framework. Our work is not innovative, but it is (we hope) a good example of a general methodology in the use of modern theorem provers.

This article is divided into three main parts: first, we introduce the basics for the λ-calculi we worked with; in the second part we describe the main features of Isabelle we used; in the third part we describe how we got the result we were speaking above.

2 The λ-Cube

We begin presenting the frame where our formalization will take place. This part is taken from [1, Chapter 5].

2.1 Pure Type Systems

A Pure Type System (PTS) is a triple: $PTS = \langle S, A, R\rangle$. S is the set of sorts; A is the set of axiom, with the form $s{:}\,t$ where $s, t \in S$; R is the set of rules, with the form $\langle s, t, r\rangle$ where $s, t, r \in S$.

Given a PTS $\langle S, A, R\rangle$, terms are defined by the following abstract grammar:

$$T = S \mid V \mid T\,T \mid \lambda V{:}\,T.T \mid \Pi V{:}\,T.T \tag{1}$$

where V is a set of variables annotated by sorts.

Let's define equality between terms: t and s are equal iff one of the following conditions holds

$$t \equiv s \tag{2}$$
$$t \equiv Z\,t' \wedge s \equiv Z\,s' \quad \wedge \quad t' = s' \tag{3}$$
$$t \equiv t'\,Z \wedge s \equiv s'\,Z \quad \wedge \quad t' = s' \tag{4}$$
$$t \equiv (\lambda x.t') \wedge s \equiv (\lambda y.s') \quad \wedge \quad t' = s'[y := x] \tag{5}$$
$$t \equiv (\lambda x.t')\,t'' \quad \wedge \quad s = t'[x := t''] \tag{6}$$

We call rule 5 α-conversion (notation $\underset{\alpha}{=}$) and rule 6 β-reduction (notation $\underset{\beta}{\rightarrow}$). The corresponding conversion relation ($\underset{\beta}{=}$) is defined as the reflective, symmetric and transitive closure of $\underset{\beta}{\rightarrow}$.

Every term has a type: we denote this fact with $t{:}\,s$, where t is the term, s is its type (but syntactically, it is a term), and : is a membership relation.

The goal of typed λ-calculus is to infer the type of a term. Inference rules have the form $\Gamma \vdash t{:}\,s$, where Γ is the *context*, i.e. the set of hypothesis, $t{:}\,s$ is the theorem and $\vdash$ is the derivation symbol.

A context Γ is recursively defined according to the following conditions:

$$\Gamma \quad = \quad \langle\rangle \tag{7}$$
$$(\Gamma \quad = \quad \Delta, x{:}\,t) \wedge (x \textit{ is a variable}) \wedge (t \textit{ is a term}) \wedge (\Delta \textit{ is a context}) \tag{8}$$

in addition we require that

$$(\Gamma = \langle\rangle, x_1{:}\,t_1, \ldots, x_n{:}\,t_n) \Rightarrow (\forall i, j.\, i \neq j \Rightarrow x_i \neq x_j) \tag{9}$$

Now is time to give inference rules: we divide them into four groups. The first one deals with context management; the second one takes terms to parts according to the principal functor; the third one contains the treatment of equality; the fourth one treats the product type (Π).

Let our PTS be $\langle Sorts, Axioms, Rules\rangle$ and let be $s \in Sorts$. The first set of rules is the following:

$$\frac{}{\langle\rangle \vdash a}\ (axiom\ [\text{proviso: } a \in Axioms]) \tag{10}$$

$$\frac{\Gamma \vdash A\colon s}{\Gamma,\ x\colon A \vdash x\colon A}\ (start) \tag{11}$$

$$\frac{\Gamma \vdash A\colon B \qquad \Gamma \vdash C\colon s}{\Gamma,\ x\colon C \vdash A\colon B}\ (weakening) \tag{12}$$

Clearly this rules are useful to manage assumptions: in a backward proof style, rule 10 is the ending point; rule 11 discards used assumptions, while rule 12 discards unused assumptions.

The second set of rules manages the principal functor of a term:

$$\frac{\Gamma \vdash C\colon A \qquad \Gamma \vdash F\colon (\Pi x\colon A.\, B)}{\Gamma \vdash F\, C\colon B[x := C]}\ (application) \tag{13}$$

$$\frac{\Gamma,\ x\colon A \vdash B\colon C \qquad \Gamma \vdash (\Pi x\colon A.\, C)\colon s}{\Gamma \vdash (\lambda x\colon A.\, B)\colon (\Pi x\colon A.\, C)}\ (abstraction) \tag{14}$$

The third set contains only one rule; its goal is to deal with equality that, in our treatment, is β-conversion:

$$\frac{\Gamma \vdash A\colon B}{\Gamma \vdash A\colon C}\ (conversion\ [\text{proviso: } B =_{\beta} C]) \tag{15}$$

And the last set is devoted to the product type:

$$\frac{\Gamma \vdash A\colon s_1 \qquad \Gamma,\ x\colon A \vdash B\colon s_2}{\Gamma \vdash (\Pi x\colon A.\, B)\colon s_3}\ (s_1, s_2, s_3\ [\text{proviso: } \langle s_1, s_2, s_3\rangle \in Rules]) \tag{16}$$

2.2 The λ-Cube

Now is time to introduce the particular typed λ-calculi we are interested in. They are referred as the λ-cube because they are partially ordered by the inclusion relation, and, graphically, the order forms a cube.

From the formal point of view, the above cited *inclusion relation* is reflected in the presence or absence of some inference rules.

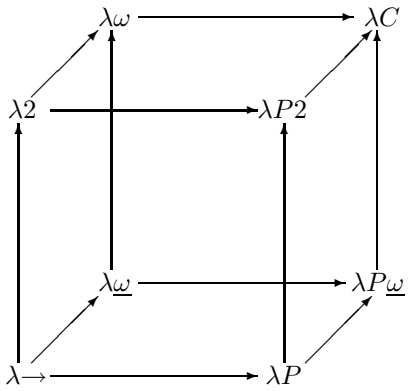

Figure 1: The λ-cube in the standard orientation

The systems in the λ-cube are formalized as PTS as follow:

$$\lambda\text{-}? = \langle \{*, \Box\}, \{*\colon \Box\}, \{\langle *, *, * \rangle\} \cup R \rangle \tag{17}$$

Each system, λ-$?$, is specified as a set of rules, R, of the form $\langle s_1, s_2, s_2 \rangle$. The eight systems in the λ-cube are:

System	Corresponding R
$\lambda\!\rightarrow$	$\emptyset$
$\lambda 2$	$\{\langle \Box, *, * \rangle\}$
λP	$\{\langle *, \Box, \Box \rangle\}$
$\lambda P2$	$\{\langle *, \Box, \Box \rangle, \langle \Box, *, * \rangle\}$
$\lambda\underline{\omega}$	$\{\langle \Box, \Box, \Box \rangle\}$
$\lambda\omega$	$\{\langle \Box, *, * \rangle, \langle \Box, \Box, \Box \rangle\}$
$\lambda P\underline{\omega}$	$\{\langle *, \Box, \Box \rangle, \langle \Box, \Box, \Box \rangle\}$
λC	$\{\langle *, \Box, \Box \rangle, \langle \Box, *, * \rangle, \langle \Box, \Box, \Box \rangle\}$

The specific rules for each system determine the way term and types may depend from each other. We will not analyze the systems and the related properties. Barendregt's works (see [1], [2]) are the right reference if you are interested in the theory. We are presenting an implementation of these ideas, so we show as much as needed in order to get our results.

However we have to note that there is an obvious strategy of proof development in some systems: if the principal functor of a term is not a product type specification then we can proceed applying the appropriate rule from the second set; if the principal functor is a variable then the only way to solve the goal is by application of rules in the first set. In other words, proof strategies for non-product terms are independent from the calculus. But we can develop deterministic proof strategies in particular calculi. In $\lambda\!\rightarrow$, for example, product type can be inferred only by the $\langle *, *, * \rangle$ rule. We will return on this point later, when we will describe tactics.

The three main problem in typed λ-calculus are type checking, typing, and type inhabiting. They are easily described; let $?X$ be an unknown λ-term, let A, B be λ-terms, and Γ a suitable context:

- Type checking means to decide whether $\Gamma \vdash A\colon B$ is true or false.
- Typing means to decide the formula $\exists ?X.\Gamma \vdash A\colon ?X$.
- Type inhabiting means to decide the formula $\exists ?X.\Gamma \vdash ?X\colon B$.

Type checking is, apparently, somewhat simpler than typing or inhabiting. We will show that carefully designed tactics can solve all these problems in the same way, using meta-variables for unknowns.

3 Isabelle

The basic Isabelle system provides a framework for developing proof systems. This framework, called *Pure* Isabelle, can be specialized for a particular logic system defining a theory, which contains rules, axioms and syntax of our logic. The *meta-logic* for these systems is the logic of Pure Isabelle, that is Intuitionistic Higher-Order Logic.

Now we will describe some relevant aspects of theories in Isabelle.

3.1 Theories

Theories are used to describe a logic, or a problem. We focus our attention to the description of logics. To describe a logic means to give a syntax, a set of axioms and a set of inference rules.

3.1.1 Syntax

Syntax is specified giving types, constants, and translations rules.

To give types we declare, in the theory specification, a `types` and an `arities` sections. For example to specify natural numbers, we could declare:

```
types
    nat 0

arities
    nat :: term
    nat :: ord
```

The first declaration means that `nat` is a type constant, which belongs to logical terms and to the set of ordered types. We can also declare type constructors, such as sets of terms:

```
types
    set 1

arities
    set :: (term) term
    set :: (term) ord
```

The meaning of these declarations is that a set of type term (whose notation is `(term) set`), is a term and it is ordered.

Constants are declared in the `consts` section of a theory file. For example, let's define some arithmetic constants:

```
consts
    "0"    :: "nat"
    succ   :: "nat => nat"
    "+"    :: "[nat, nat] => nat"    ("_ + _" [0 ,1] 0)
```

We say that 0 is constant, and its type is `nat`, i.e. it is a natural number; *succ* is a function from naturals to naturals; + is a function from a pair of natural numbers to naturals, and its syntax is not $+(a, b)$, but $a + b$, and it associate on the left, i.e. $a + b + c$ is parsed as $(a + b) + c$.

Mixfix declarations (like the syntax of +) are not enough powerful to specify complex syntaxes, so we need translations rules. For example, let's consider the universal quantifier over natural numbers:

```
consts
    forall :: "(nat => bool) => bool"
    "@all" :: "[id, nat] => bool"        ("(ALL _ . _)")

translations
    "(ALL x . E(x))" == "forall(%x.E(x))"
```

The constant `forall` takes as argument a function from natural numbers to booleans, and the intended meaning is that, if this function, let's say f, is everywhere true, then *forall* (f) is true, and it is false otherwise. To setup the syntax properly, we define another costant, `@all`, that takes as arguments an identifier and an expression of type `nat`; then the parser translates this notation to *forall* $(\lambda x.E)$, and then it accepts such an expression because it agrees with the syntax of the `forall` declaration. The printing process reverse the effect of our translation, so every term whose principal functor is `forall` is written out according to the syntax of `@all`.

3.1.2 Axioms and Inference Rules

Axioms and inference rules are specified as meta-level expressions.

As an example, let's give some axioms for the natural numbers:

```
rules
    succ_inj    "succ x = succ y --> x = y"
    succ_zero   "ALL x. ~(succ x = 0)"
```

We have used some predefined constants, such as `ALL` and `-->`, along with their syntaxes. Let's now give an inference rule, for example the induction over natural numbers:

```
rules
    induct    "[| P(0); (P(x) ==> P(succ x))|] ==> (ALL x. P(x))"
```

The meaning of meta-level implication (`==>`), is "from the assumption(s), we can prove". So we can read the induction rule as: "from the assumption $P(0)$ and, if we can prove $P(succ\,x)$ from $P(x)$, then is proved $\forall x.P(x)$.

3.2 Proofs

Proofs in Isabelle consist in setting a goal and applying inference rules to reduce goals into subgoals or to prove a subgoal.

Inference rules, being nothing more than meta-level expressions, cannot be applied directly, but they are *resolved* against the goal.

For example, given the induction rule, we want to prove $\forall x.(x = 0) \vee (\exists y.x = succ\,(y))$. We setup the previous formula as the goal, then we match the conclusion of the induction rule with the goal, obtaining two subgoals: $(0 = 0) \vee (\exists y.0 = succ\,(y))$ and $(succ\,(x) = 0) \vee (\exists y.succ\,(x) = succ\,(y))$, given the assumtion $\forall x.(x = 0) \vee (\exists y.x = succ\,(y))$. The first subgoal is true, because $0 = 0$, and the second is true, making x and y equal.

Resolution is not the only way to use inference rules, in fact, we can use definitions (expressions of the form `A == B`), to rewrite our terms, we can simplify expressions using conversion rules, and so more.

We call *tactics* all of these proof methods.

3.3 Tactics and Tacticals

Inference rules of object-logics are meta-theorems. To avoid confusion, and to develop an higher level of understanding of proving techniques, we call *tactics* the inference rules of the meta-level. Some tactics are primitives, i.e. they are not derived, nor they are derivable. For example, resolution is a primitive tactic. Other tactics are complex, and their behaviour is the sum of lots of primitive inferences composed in a proper way. Simplification is an example of such a tactic.

When we develop a new theory, an obvious step is to set up some proof methods. For example, Peano arithmetic is defined when we give the syntax of arithmetic expression and the axioms. But arithmetic can be effectively used only when we develop an inference rule for induction; and we can use properly Peano arithmetic only when we have a tactic to automate the application of such a rule.

Syntax	Behaviour
THEN	Sequence
ORELSE	Deterministic choice
REPEAT	Non-deterministic repetition
DEPTH_FIRST	Depth first searching strategy
BREADTH_FIRST	Breadth first searching strategy
BEST_FIRST	Heuristic searching strategy
FILTER	Filter goals before applying tactic

Table 1: Some common tacticals

Proof methods can be thought as specialized inference rules, but also as proof strategies. Strategies can be thought as inference rules, performing "long" steps in the development of a proof.

Strategies are not fundamental because everything we can do with them, in principle, can be done without. But they are the instrument to implement in a modern theorem prover the idea of "obviously". Often, when we look at an handwritten proof, we see "the proof of the last assertion is trivial"; startegies are the way to perform such steps, without expanding a mechanical proof. Strategies are also a method to search for proofs, using a more refined approach than the try-all-possibilities method.

In both cases, strategies are implemented combining the existing tactics in some way to produce a new, well-formed tactic. To combine tactics we use some operators, called *tacticals*, that guarantee to preserve soundness.

Some common and useful tacticals are listed in table 1.

4 Our Implementation

Now let's see how to implement the λ-cube as an Isabelle's theory.

Our approach is standard: we begin with the definition of types and sorts, then we setup the syntax properly, and finally, the inference rules. With the theory so defined, we can start to develop tactics and theorems.

4.1 Theory Definition

Our theory will be named `Lambda`, and it will be based over the most basic theory of Isabelle, `Pure`.

```
Lambda = Pure +
...
```

4.1.1 Syntax

We need three different types: terms, contexts, and typings, i.e. a couple of terms. The only "true" type is term, the other ones are useful only for syntactical purposes. So we declare `term` a subtype of `logic`, inheriting what `Pure` offers for the treatment of this type.

```
...
types
  term  context  typing
arities
  term :: logic
...
```

The syntax of a derivation is:

$$\mathit{Derivation} = \mathit{Context} \vdash \mathit{Typing} \tag{18}$$

When *Context* is void we admit a shorter syntax form for 18:

$$\mathit{ShortDerivation} = \mathit{Typing}$$

which is equivalent to $\mathit{EmptyContext} \vdash \mathit{Typing}$. The resulting declarations to Isabelle are:

```
...
consts
  Trueprop      :: "[context, typing] => prop"    ("(_/ |- _)")
  Trueprop1     :: "typing => prop"               ("(_)")
  ...
translations
  (prop) "x:X"  == (prop) "|- x:X"
  ...
```

The type of the derivation syntactical structure is `prop`, because its semantic value is a meta-level truth value. So, `Trueprop` is used also to reflect object-level truth into the meta-level.

The syntax of typings and contexts are:

$$\mathit{Typing} = \mathit{Term} : \mathit{Term} \tag{19}$$

$$\begin{aligned} \mathit{Context} = \quad & \mathit{EmptyContext} \quad | \\ & \textbf{Meta-Variable} \quad | \\ & \textbf{Meta-Identifier} \quad | \\ & \mathit{Context}\ \ \mathit{Typing} \end{aligned} \tag{20}$$

We allow a bit of syntactical sugar: the empty context can be written as <>. The implementation of such a grammar is straightforward:

```
...
consts
  ...
  Has_type      :: "[term, term] => typing"       ("(_:_)")

  MT_context    :: "context"                      ("")
  Void_context  :: "context"                      ("<>")
  ""            :: "id => context"                ("_ ")
  ""            :: "var => context"               ("_ ")
  Context       :: "[context, typing] => context" ("_ _")
  ...
translations
  ...
  (context)"<>" => (context)""
  ...
```

The translation rule is one-way; we can write the empty context as <>, but it is always printed as the null string.

The syntax of λ-terms is:

$$\begin{aligned} \mathit{Term} = \quad & * \quad | \\ & \Box \quad | \\ & \lambda\, \mathit{Variable}\colon \mathit{Term}.\mathit{Term} \quad | \\ & \Pi\, \mathit{Variable}\colon \mathit{Term}.\mathit{Term} \end{aligned} \tag{21}$$

To implement bindings in λ and Π constructions, we use two internal representations, with higher order functions:

$$\Pi x\colon A.B = \mathit{Prod}(A, \lambda x.B) \tag{22}$$

$$\lambda x\colon A.B = \mathit{Abs}(A, \lambda x.B) \tag{23}$$

Here the λ to the left is typed, while the one to the right is untyped, i.e. a meta-level abstraction operator.

We allow the usual definition of the free product type:

$$A \to B = \Pi x\colon A.B \qquad , x \notin \mathit{FreeVars}(B) \tag{24}$$

With these premises, our implementation of the syntax of terms is:

```
...
consts
  ...
  star          :: "term"                         ("*")
  box           :: "term"                         ("[]")
  "^"           :: "[term, term] => term"         (infixl 20)
  Abs, Prod     :: "[term, term => term] => term"
```

```
  Lam           :: "[idt, term, term] => term"
                                ("(3Lam _:_./ _)" [0,0] 10)
  Pi            :: "[idt, term, term] => term"
                                ("(3Pi _:_./ _)"  [0,0] 10)
  "->"          :: "[term, term] => term"         (infixr 10)
translations
  ...
  "Lam x:A. B"  == "Abs(A, %x. B)"
  "Pi x:A. B"   => "Prod(A, %x. B)"
  "A -> B"      => "Prod(A, _K(B))"
...
ML
val print_translation = [("Prod",dependent_tr'("Pi","op->"))];
```

4.1.2 Axioms and Inference Rules

Now we present inference rules of our object-level. To simplify the theory, we define only λC, and we distinguish among the other theories in tactics.

```
...
rules
  axiom         "       *:[]"
  start_s       "       G |- A:* ==> \
\                       G x:A |- x:A"
  start_b       "       G |- A:[] ==> \
\                       G x:A |- x:A"
  weak_s        "[|     G |- B:C;        \
\                       G |- A:*  |] ==> \
\                       G x:A |- B:C"
  weak_b        "[|     G |- B:C;        \
\                       G |- A:[] |] ==> \
\                       G x:A |- B:C"

  app           "[|     G |- C:A;                \
\                       G |- F:Prod(A, B) |] ==> \
\                       G |- F^C: B(C)"
  abs_s         "[|     !!x. G x:A |- B(x):C(x); \
\                       G |- Prod(A,C):*  |] ==> \
\                       G |- Abs(A,B):Prod(A,C)"
  abs_b         "[|     !!x. G x:A |- B(x):C(x); \
\                       G |- Prod(A,C):[] |] ==> \
\                       G |- Abs(A,B):Prod(A,C)"
  conv_s        "[|     G |- A:B;     \
\                       G |- C:*;     \
\                       C == B |] ==> \
\                       G |- A:C"
```

```
  conv_b        "[|     G |- A:B;     \
\                       G |- C:[];    \
\                       C == B |] ==> \
\                       G |- A:C"

  rule_ss       "[|     G |- A:*;                    \
\                       !!x. G x:A |- B(x):* |] ==> \
\                       G |- Prod(A, B):*"
  rule_bs       "[|     G |- A:[];                   \
\                       !!x. G x:A |- B(x):* |] ==> \
\                       G |- Prod(A, B):*"
  rule_sb       "[|     G |- A:*;                     \
\                       !!x. G x:A |- B(x):[] |] ==> \
\                       G |- Prod(A, B):[]"
  rule_bb       "[|     G |- A:[];                    \
\                       !!x. G x:A |- B(x):[] |] ==> \
\                       G |- Prod(A, B):[]"

  beta          "Abs(A, f)^a == f(a)"
end
...
```

In the next section we will discuss the use of these rules, but now we want to point out the way conversion rules and $\underset{\beta}{=}$ are implemented.

The intended meaning of $\underset{\beta}{=}$ is intensional equality, i.e. two λ-expressions are equal iff they denote the same "function". Axioms governing such a relation are reflexivity, symmetry, transitivity, closure under application (both left and right), and abstraction, α-reduction and β-reduction. The meta-level equality is already defined to enjoy all these properties, except β-reduction, and so the need of the axiom `beta`.

Conversion rules substitute the $\underset{\beta}{=}$ relation with meta-level equality. From an operational point of view, the application of a conversion rule implies that we have to prove that two terms are equal in the meta-level.

We have to note a little thing: ordering of hypothesis in inference rules is not important from a logical point of view, but from an operational point of view it is relevant; if we assume to try solve every time the first subgoal, the ordering we have given to assumptions guarantees that the smallest number of unknowns is generated, specially when we have to solve a typing or a type inhabiting problem.

4.2 Specialized Tactics

What we need now, is to develop some tactics to handle obvious proofs in a standard way.

To manage contexts we need the rules 10, 11 and 12. We write two low-level tactics, `context_s_step_tac` and `context_b_step_tac`, to treat contexts using

the $*$ or the $\Box$ version of inference rules.

```
open Lambda;

fun context_s_step_tac n =
    (resolve_tac [axiom] n)   ORELSE
    (resolve_tac [start_s] n) ORELSE
    (resolve_tac [weak_s] n);

fun context_b_step_tac n =
    (resolve_tac [axiom] n)   ORELSE
    (resolve_tac [start_b] n) ORELSE
    (resolve_tac [weak_b] n);
...
```

To manage contexts we can write a tactic which, by backtracking, may apply both of these low-level tactics.

```
...
fun context_step_tac n = (context_s_step_tac n) APPEND
                         (context_b_step_tac n);
...
```

But, it is easy to prove that `context_b_step_tac` is the only admissible tactic to apply when we have to prove $\Gamma, x{:}* \vdash y{:}A$. This consideration leads to the following tactic:

```
...
fun context_step_fun (Const("all", _) $ Abs(_, _, x), n) =
    context_step_fun (x, n)
  | context_step_fun (Const("Trueprop", _) $ x $ _, n) =
    context_step_fun (x, n)
  | context_step_fun (Const("Context", _) $ _ $ x, n) =
    context_step_fun (x, n)
  | context_step_fun (Const("Has_type", _) $ _ $
                      Const("star", _), n) =
    context_b_step_tac n
  | context_step_fun (_, n) =
    (context_s_step_tac n) APPEND
    (context_b_step_tac n);

fun context_step_tac n = SUBGOAL context_step_fun n;
...
```

We note that, in the particular cases of $\lambda{\to}$ and $\lambda 2$, we can develop a deterministic tactic to treat contexts. In fact, it is easy to prove that, when the goal has the form $\Gamma, x{:}A \vdash B{:}C$, and $A \neq *$, the only admissible tactic is `context_s_step_tac`. This leads to:

```
...
fun special_context_step_fun
                       (Const("all", _) $ Abs(_, _, x), n) =
    special_context_step_fun (x, n)
  | special_context_step_fun (Const("Trueprop", _) $ x $ _, n) =
    special_context_step_fun (x, n)
  | special_context_step_fun (Const("Context", _) $ _ $ x, n) =
    special_context_step_fun (x, n)
  | special_context_step_fun (Const("Has_type", _) $ _ $
                              Const("star", _), n) =
    context_b_step_tac n
  | special_context_step_fun (_, n) =
    context_s_step_tac n;

fun special_context_step_tac n =
                SUBGOAL special_context_step_fun n;
...
```

Here we want to develop a prover for $\lambda\rightarrow$. Let's work top-down: let `simply_type_tac` be our prover, it has to repeat proving steps until there are no more goals to solve. So we have to develop a tactic to choose the right proof step: let it be `operand_ss_step_tac`. Now we need to develop an algorithm to choose the right inference rule to apply to a goal.

Looking at the inference rules, we note that, given a term, only a small set of rules can be applied to. More specifically, if we have to prove $\Gamma \vdash A\colon B$, we can apply the following rules, according the principal functor of term A:

Functor	**Tactics**
Variable	special_context_step_tac
Application	resolve_tac [app, conv_s]
Abstraction	resolve_tac [abs_s, conv_s]
Product	resolve_tac [rule_ss, conv_s]

If we ignore the conversion rule, not so useful in $\lambda\!\!\rightarrow$, we obtain a deterministic algorithm to choose the tactic to apply to a goal. Writing down these things in SML code, we obtain:

```
...
fun operand_step_fun (Const("all", _) $ Abs(_, _, x), n) =
    operand_step_fun (x, n)
  | operand_step_fun ((Const("Trueprop", _) $ _ $
                       (Const("Has_type", _) $
                        (Const("^", _) $ _ $ _) $ _)), n) =
    (resolve_tac [app] n)
  | operand_step_fun ((Const("Trueprop", _) $ _ $
                       (Const("Has_type", _) $
                        (Const("Abs", _) $ _ $ _) $ _)), n) =
```

```
    (resolve_tac [abs_s] n)
  | operand_step_fun ((Const("Trueprop", _) $ _ $
                       (Const("Has_type", _) $
                        (Const("Prod", _) $ _ $ _) $ _)), n) =
    (resolve_tac [rule_ss] n)
  | operand_step_fun (x, n) =
    (context_step_fun (x, n));

fun operand_ss_step_tac n =
        SUBGOAL (fn (x, i) =>
                 operand_step_fun (x, i)) n;

val simply_typed_tac = REPEAT_FIRST operand_ss_step_tac;
```

5 Conclusions

Typed λ-calculus can be automated in Isabelle in a quite natural way. Our implementation of those theories is not very efficient: we could develop, for example, variants of the inference rules (the primitive ones) to avoid repetition in proofs, or we could develop tactics to maintain a database of proven facts during the proof process to avoid duplication of proofs.

As we said in the introduction, this is a case study in the use of Isabelle: we are not interested in "the best way" to implement our theory, but in "the most natural" one.

We think that our work is a good example to show how to use Isabelle to implement a mathematical theory. It is *small*, because we have a little number of rules, and a real syntax, although not very complex. It is *clean*, because the mathematical theory of λ-calculus is so, and our implementation is direct.

There are some problems: the first one is absence of semantic. It is possible to modify slightly our syntax for compatibility with `ZF`, and to develop, under `ZF`, a function mapping from λ-terms to domains. Another problem is the absence of data-types, and other useful applications of λ-calculus.

Currently we are working on the implementation of Böhm–Berarducci algorithm to represent data structures in $\lambda 2$.

Perhaps in the near future we will work on enhancements to the theory we have presented, and we hope to develop a semantic in `ZF`.

Acknowledgments

This work was developed in the Computational Architectures Laboratory at Computer Science Department – State University of Milan (Italy). Essential support was given by Professor Giovanni Degli Antoni.

References

[1] H. P. Barendregt. *Lambda Calculi with Types*, in *Handbook of Logic in Computer Science*, Vol. II edited by S. Abramsky, D. Gabbay and T. S. E. Maibaum, Oxford University Press (preprint 1992).

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, revised edition. Studies in Logic and the Foundations of Mathematics, North Holland (1984).

[3] Sara Kalvala. *A Gentle Introduction to Isabelle*, University of Cambridge, Computer Laboratory.

[4] Lawrence Paulson. *Introduction to Isabelle*, University of Cambridge, Computer Laboratory.

[5] Lawrence Paulson. *The Isabelle Reference Manual*, University of Cambridge, Computer Laboratory.

[6] Lawrence Paulson. *Isabelle's Objects-Logics*, University of Cambridge, Computer Laboratory.

[7] Lawrence Paulson. *ML for the Working Programmer*, Cambridge University Press (1991)

A Theory File

Here is the text of the `Lambda.thy` file:

```
Lambda = Pure +
types
  term  context  typing
arities
  term :: logic
consts
  Trueprop      :: "[context, typing] => prop"    ("(_/ |- _)")
  Trueprop1     :: "typing => prop"               ("(_)")
  Has_type      :: "[term, term] => typing"       ("(_:_)")

  MT_context    :: "context"                      ("")
  Void_context  :: "context"                      ("<>")
  ""            :: "id => context"                ("_ ")
  ""            :: "var => context"               ("_ ")
  Context       :: "[context, typing] => context" ("_ _")
  star          :: "term"                         ("*")
  box           :: "term"                         ("[]")
  "^"           :: "[term, term] => term"         (infixl 20)
  Abs, Prod     :: "[term, term => term] => term"
```

```
  Lam           :: "[idt, term, term] => term"
                                ("(3Lam _:_./ _)" [0,0] 10)
  Pi            :: "[idt, term, term] => term"
                                ("(3Pi _:_./ _)"  [0,0] 10)
  "->"          :: "[term, term] => term"         (infixr 10)
translations
  (prop) "x:X"  == (prop) "|- x:X"
  (context)"<>" => (context)""
  "Lam x:A. B"  == "Abs(A, %x. B)"
  "Pi x:A. B"   => "Prod(A, %x. B)"
  "A -> B"      => "Prod(A, _K(B))"
rules
  axiom         "       *:[]"
  start_s       "       G |- A:* ==> \
\                       G x:A |- x:A"
  start_b       "       G |- A:[] ==> \
\                       G x:A |- x:A"
  weak_s        "[|     G |- B:C;        \
\                       G |- A:*  |] ==> \
\                       G x:A |- B:C"
  weak_b        "[|     G |- B:C;        \
\                       G |- A:[] |] ==> \
\                       G x:A |- B:C"

  app           "[|     G |- C:A;                \
\                       G |- F:Prod(A, B) |] ==> \
\                       G |- F^C: B(C)"
  abs_s         "[|     !!x. G x:A |- B(x):C(x); \
\                       G |- Prod(A,C):*  |] ==> \
\                       G |- Abs(A,B):Prod(A,C)"
  abs_b         "[|     !!x. G x:A |- B(x):C(x); \
\                       G |- Prod(A,C):[] |] ==> \
\                       G |- Abs(A,B):Prod(A,C)"
  conv_s        "[|     G |- A:B;     \
\                       G |- C:*;     \
\                       C == B |] ==> \
\                       G |- A:C"
  conv_b        "[|     G |- A:B;     \
\                       G |- C:[];    \
\                       C == B |] ==> \
\                       G |- A:C"

  rule_ss       "[|     G |- A:*;                    \
\                       !!x. G x:A |- B(x):* |] ==> \
\                       G |- Prod(A, B):*"
  rule_bs       "[|     G |- A:[];                  \
```

```
\                       !!x. G x:A |- B(x):* |] ==> \
\                       G |- Prod(A, B):*"
  rule_sb       "[|     G |- A:*;                    \
\                       !!x. G x:A |- B(x):[] |] ==> \
\                       G |- Prod(A, B):[]"
  rule_bb       "[|     G |- A:[];                   \
\                       !!x. G x:A |- B(x):[] |] ==> \
\                       G |- Prod(A, B):[]"

  beta          "Abs(A, f)^a == f(a)"
end

ML
val print_translation = [("Prod",dependent_tr'("Pi","op->"))];
```

B ML File

Here is the text of the `Lambda.ML` file:

```
open Lambda;

(* Context management tactics *)

fun context_s_step_tac n =
    (resolve_tac [axiom] n)   ORELSE
    (resolve_tac [start_s] n) ORELSE
    (resolve_tac [weak_s] n);

fun context_b_step_tac n =
    (resolve_tac [axiom] n)   ORELSE
    (resolve_tac [start_b] n) ORELSE
    (resolve_tac [weak_b] n);

fun context_step_fun (Const("all", _) $ Abs(_, _, x), n) =
    context_step_fun (x, n)
  | context_step_fun (Const("Trueprop", _) $ x $ _, n) =
    context_step_fun (x, n)
  | context_step_fun (Const("Context", _) $ _ $ x, n) =
    context_step_fun (x, n)
  | context_step_fun (Const("Has_type", _) $ _ $
                      Const("star", _), n) =
    context_b_step_tac n
  | context_step_fun (_, n) =
    (context_s_step_tac n) APPEND
    (context_b_step_tac n);
```

```
fun context_step_tac n = SUBGOAL context_step_fun n;

fun special_context_step_fun
                       (Const("all", _) $ Abs(_, _, x), n) =
    special_context_step_fun (x, n)
  | special_context_step_fun (Const("Trueprop", _) $ x $ _, n) =
    special_context_step_fun (x, n)
  | special_context_step_fun (Const("Context", _) $ _ $ x, n) =
    special_context_step_fun (x, n)
  | special_context_step_fun (Const("Has_type", _) $ _ $
                              Const("star", _), n) =
    context_b_step_tac n
  | special_context_step_fun (_, n) =
    context_s_step_tac n;

fun special_context_step_tac n =
               SUBGOAL special_context_step_fun n;

(* Simple Typed Tactic *)

fun operand_step_fun (Const("all", _) $ Abs(_, _, x), n) =
    operand_step_fun (x, n)
  | operand_step_fun ((Const("Trueprop", _) $ _ $
                      (Const("Has_type", _) $
                       (Const("^", _) $ _ $ _) $ _)), n) =
    (resolve_tac [app] n)
  | operand_step_fun ((Const("Trueprop", _) $ _ $
                       (Const("Has_type", _) $
                        (Const("Abs", _) $ _ $ _) $ _)), n) =
    (resolve_tac [abs_s] n)
  | operand_step_fun ((Const("Trueprop", _) $ _ $
                       (Const("Has_type", _) $
                        (Const("Prod", _) $ _ $ _) $ _)), n) =
    (resolve_tac [rule_ss] n)
  | operand_step_fun (x, n) =
    (context_step_fun (x, n));

fun operand_ss_step_tac n =
        SUBGOAL (fn (x, i) =>
                 operand_step_fun (x, i)) n;

val simply_typed_tac = REPEAT_FIRST operand_ss_step_tac;
```