6

A simple type-theoretic language: Mini-TT

Thierry Coquand
Chalmers University of Technology and Göteborg University

Yoshiki Kinoshita
National Institute of Advanced Industrial Science and Technology (AIST), Japan

Bengt Nordström
Chalmers University of Technology and Göteborg University

Makoto Takeyama
National Institute of Advanced Industrial Science and Technology (AIST), Japan

Abstract

This paper presents a formal description of a small functional language with dependent types. The language contains data types, mutual recursive/inductive definitions and a universe of small types. The syntax, semantics and type system is specified in such a way that the implementation of a parser, interpreter and type checker is straightforward. The main difficulty is to design the conversion algorithm in such a way that it works for open expressions. The paper ends with a complete implementation in Haskell (around 400 lines of code).

6.1 Introduction

We are going to describe a small language with dependent types, its syntax, operational semantics and type system. This is in the spirit of the paper "A simple applicative language: Mini-ML" by Clément, Despeyroux, and Kahn [5], where they explain a small functional language. From them we have borrowed the idea of using patterns instead of variables in abstractions and let-bindings. It gives an elegant way to express mutually recursive definitions. We also share with them the view that a programming language should not only be formally specified, but it should also be possible to reason about the correctness of its implementation. There should be a small step from the formal operational semantics to an interpreter and also between the specification of the type system to a type checker.

Our type checking algorithm reduces the problem to checking convertibility of terms[1]. A central feature of our Mini-TT presentation is that we compute normal forms of open terms for convertibility checking.

A major problem has been to define the computation of normal forms in such a way that checking convertibility can be reduced to checking syntactic identity of the normal forms. This is done in two steps: first evaluating an expression to its value and then applying a readback function taking the value to a normal expression. Values are representations of expressions in weak head normal form. There are connections between our work and the work of B. Gregoire and X. Leroy on compilation of strong reductions [10, 9]. Like in their work, our approach for conversion is based on weak reductions on open terms, complemented by a recursive "read back" procedure.

Two main differences with this work are the following. First, there is the use of patterns mentioned above to encode mutual recursive definitions and our representation of data types as labelled sums. These two features allow us to represent some inductive recursive definitions in a natural way. Second, our way of comparing functions defined by case is less refined than the one in [10]. Though less refined, we think that our approach is simpler to implement at a machine level. The programming language associated to type theory is usually presented as λ-calculus extended with some primitive constants (constructors) and constants implicitly defined by pattern-matching equations [13]. Our simple treatment is actually more faithful to this usual presentation than structural equality. It has also the advantage that our syntax is very close to the one of an ordinary functional programming language, with arbitrary mutual recursive definitions.

Our approach should also allow us to apply the results of [4, 7]. This should provide a modular and semantical sufficient condition ensuring strong normalisation and hence decidability of our type-checking algorithm.

This paper is organised as follows. In Section 6.2, the syntax of Mini-TT is given, as well as some syntactic sugar. Some programming examples, such as booleans and natural numbers, are given in Section 6.3. Section 6.4 introduces values and the evaluation function that sends expressions to values. Our semantics is not based on a reduction rela-

[1] Conversely, the convertibility of two terms t, u of type A can be reduced to the problem of whether the term $\lambda T.\lambda x.x$ has type $\Pi T : A \to \mathsf{U}.\ T\ t \to T\ u$.

expressions	M, N, A, B	$::=$	$\lambda p \,.\, M \mid x \mid M\,N \mid \Pi p : A \,.\, B \mid \mathsf{U} \mid$
			$M, N \mid M.1 \mid M.2 \mid \Sigma p : A \,.\, B \mid$
			$0 \mid \mathbf{1} \mid$
			$c\,M \mid \mathsf{fun}\,S \mid \mathsf{Sum}\,S \mid$
			$D; M$
patterns	p	$::=$	$x \mid p, p \mid \_$
choices	S	$::=$	$() \mid (c\;M, S)$
declarations	D	$::=$	$p : A = M \mid \mathsf{rec}\,p : A = M$
syntactic sugar	$A \to B$	$=$	$\Pi \_ : A \,.\, B$
	$A \times B$	$=$	$\Sigma \_ : A \,.\, B$
	$c\,A \mid S$	$=$	$(c, A), S$
	$c \to M \mid S$	$=$	$(c, M), S$

Fig. 6.1. Syntax of Mini-TT

tion between expressions. Intuitively, a value represents an expression in weak head normal form and the evaluation function implements the weak head reduction. Section 6.5 defines normal expressions and the readback function that sends a value to a normal expression. We check the convertibility of expressions by first evaluating them to values, then applying the readback function, and finally checking for syntactic identity. Typing rules are presented in Section 6.6, and Section 6.8 discusses variations possibly applied to mini-TT as is given here. Finally Section 6.9 concludes the paper. In the appendix, we attach a Haskell code which checks the typing relation, i.e, given two expressions, checks whether the latter is a type expression and the former has the latter as its type.

6.2 Syntax

A brief summary of the syntax can be found in figure 6.1.

In this presentation of the language, we are using patterns to introduce variables. An abstraction of the form $\lambda\,(x, y) \,.\, e$ is an abstraction of two variables x and y, so $(\lambda\,(x, y) \,.\, e)\; u$ reduces to $e[x := u.1, y := u.2]$ while an abstraction of the form $\lambda\,\_ \,.\, e$ is an abstraction of no variables, so $(\lambda\,\_ \,.\, e)\; u$ reduces to e.

A program is an expression of type $\mathbf{1}$, usually just a list of declarations. A declaration is a definition of a constant with its type. We will first explain the syntax of the declarations, then continue to describe the various ways of forming expressions associated to each type forming operation (unit type, dependent product, labelled sum and universe).

Declarations: Recursive and explicit definitions There are two kinds of definitions, let expression $p : A = M;\ N$, and letrec expression $\mathsf{rec}\, p : A = M;\ N$. Use of patterns is not strictly necessary but simplifies the definition of mutually recursive definitions.

We allow definitions of non-terminating functions. This is essential if Mini-TT is going to be a core language for programming. Non-terminating functions are essential for interactive programs. Of course, it causes problems for type-checking to be terminating, so we assume that termination is checked in a separate phase.

Unit type The unit type $\mathbf{1}$ has the unit element 0.

Dependent product, lambda abstraction, application The dependent product type $\Pi\, p : A \,.\, B$ is the type of functions taking an object M in the type A to an object in the type B (where B may depend on M). Lambda abstractions are written $\lambda\, p \,.\, M$ and application $M\, N$. It is possible to use the notation $A \to B$ as syntactic sugar for $\Pi\, \_ : A \,.\, B$.

Dependent sum, pairs and projections The dependent sum $\Sigma\, p : A \,.\, B$ is the type of pairs M, N, where $M \in A$, $N \in B[M/p]$. The projections are written $M.1$ and $M.2$. It is possible to use the notaton $A \times B$ as syntactic sugar for $\Sigma\, \_ : A \,.\, B$.

Labelled sum, constructor application and case An inductive set is looked as a labelled sum $\mathsf{Sum}(c_1\, A_1, \ldots, c_n\, A_n)$, which contains objects of the form $c_i\, E$, where E is an object in A_i. We will also write this as $\mathsf{Sum}(c_1\, A_1 \mid \cdots \mid c_n\, A_n)$. It is possible to skip the type A_i in the case that it is the unit type $\mathbf{1}$. For instance, the type of Boolean values can be written as $\mathsf{Sum}\ (\mathsf{true} \mid \mathsf{false})$ instead of $\mathsf{Sum}\ (\mathsf{true}\ \mathbf{1} \mid \mathsf{false}\ \mathbf{1})$.

The case-analysis function has the shape $\mathsf{fun}(c_1\ M_1, \ldots, c_n\ M_n)$. It is a function which when applied to an object of the form $c_i\, N$ is equal to $M_i\, N$. The choice $c_i\ (\lambda\, p \,.\, M_i)$ is written $c_i\ p \to M_i$ and the choice $c_i\ (\lambda\, \_ \,.\, M_i)$ is written $c_i \to M_i$.

Universe The type of small types is written U. The objects in this are types not built up using U.

6.3 Examples of programs

Here are some examples of programs (a list of declarations $D_1; \cdots ; D_n$) that we can write in Mini-TT. The generic identity function will be represented by the program

$$\mathsf{id} : \Pi\, A : \mathsf{U}\, .\, A \to A = \lambda\, A\, .\, \lambda\, x\, .\, x$$

A simple example is the data type of Booleans and the corresponding elimination function:

$$\begin{array}{lcl} \mathsf{Bool} & : & \mathsf{U} = \mathsf{Sum}\ (\mathsf{true} \mid \mathsf{false}) \\ \mathsf{elimBool} & : & \Pi\, C : \mathsf{Bool} \to \mathsf{U}\, .\, C\,\mathsf{false} \to C\,\mathsf{true} \to \Pi\, b : \mathsf{Bool}\, .\, C\, b \\ & = & \lambda\, C\, .\, \lambda\, h_0\, .\, \lambda\, h_1\, .\, \mathsf{fun}\ (\mathsf{true} \to h_1 \mid \mathsf{false} \to h_0) \end{array}$$

The type of natural numbers is represented as a recursively defined labelled sum

$$\mathsf{rec}\ \ \mathsf{Nat} : \mathsf{U} = \mathsf{Sum}\ (\mathsf{zero} \mid \mathsf{succ}\ \mathsf{Nat})$$

Similarly, the type of lists is described by

$$\mathsf{rec}\ \ \mathsf{List} : \mathsf{U} \to \mathsf{U} = \lambda\, A\, .\, \mathsf{Sum}\ (\mathsf{nil} \mid \mathsf{cons}\, A \times \mathsf{List}\, A)$$

The elimination function of the type of natural numbers is the recursively defined function

$$\begin{array}{lcl} \mathsf{rec}\ \ \mathsf{natrec} & : & \Pi\, C : \mathsf{Nat} \to \mathsf{U}\, .\, C\,\mathsf{zero} \to (\Pi\, n : \mathsf{Nat}\, .\, C\, n \to C(\mathsf{succ}\, n)) \to \\ & & \Pi\, n : \mathsf{Nat}\, .\, C\, n \\ & = & \lambda\, C\, .\, \lambda\, a\, .\, \lambda\, g\, .\, \mathsf{fun}\ (\mathsf{zero} \to a \mid \mathsf{succ}\, n_1 \to g\ n_1\ (\mathsf{natrec}\ C\, a\, g\, n_1)) \end{array}$$

If we work in this fragment, and we do not introduce new definitions using rec, Sum and fun, we obtain a faithful representation of the corresponding fragment of type theory described in Chapter 20 of [12].

In Mini-TT, we can directly introduce other recursive functions on natural numbers, even if they can be defined without recursion using natrec. A simple example is the addition function

$$\begin{array}{lcl} \mathsf{rec}\ \ \mathsf{add} & : & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ & = & \lambda\, x\, .\, \mathsf{fun}\ (\mathsf{zero} \to x \mid \mathsf{succ}\ y_1 \to \mathsf{succ}\, (\mathsf{add}\, x\, y_1)) \end{array}$$

A more complex example is provided by the decidable equality function

$$\begin{array}{lcl} \mathsf{rec}\ \ \mathsf{eqNat} & : & \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Bool} \\ & = & \mathsf{fun}\ (\,\mathsf{zero} \quad\ \to \mathsf{fun}\ (\mathsf{zero} \to \mathsf{true} \mid \mathsf{succ}\ y \to \mathsf{false}) \\ & & \qquad \mid \mathsf{succ}\ x \to \mathsf{fun}\ (\mathsf{zero} \to \mathsf{false} \mid \mathsf{succ}\ y \to \mathsf{eqNat}\, x\, y)) \end{array}$$

$$\begin{array}{rlcl}
\text{values} & u, v, t & ::= & [k] \mid \lambda f \mid \Pi\, t\, g \mid \mathsf{U} \mid \\
 & & & u, v \mid 0 \mid \Sigma\, t\, g \mid \mathbf{1} \mid \\
 & & & c\, v \mid \mathsf{fun}\, s \mid \mathsf{Sum}\, s
\end{array}$$

$$\begin{array}{rlcl}
\text{neutral values (accumulators)} & k & ::= & \mathsf{x}_n \mid k\, v \mid k.1 \mid k.2 \mid s\, k \\
\text{function closures} & f, g & ::= & \langle \lambda p.M, \rho \rangle \mid f \circ c \\
\text{choice closures} & s & ::= & \langle S, \rho \rangle \\
\text{environments} & \rho & ::= & () \mid \rho, p = v \mid \rho, D
\end{array}$$

Fig. 6.2. Values

Our representation of this function corresponds to the system of pattern-matching equations

$$\begin{array}{llll}
\mathsf{eqNat\ zero} & \mathsf{zero} = \mathsf{true}, & \mathsf{eqNat\ zero} & (\mathsf{succ}\ y) = \mathsf{false}, \\
\mathsf{eqNat}\ (\mathsf{succ}\ x) & \mathsf{zero} = \mathsf{false}, & \mathsf{eqNat}\ (\mathsf{succ}\ x) & (\mathsf{succ}\ y) = \mathsf{eqNat}\ x\ y,
\end{array}$$

compiled using two auxiliary functions

$$\begin{array}{lll}
\mathsf{eqNat\ zero} \quad = f, & f\ \mathsf{zero} = \mathsf{true}, & f\ (\mathsf{succ}\ y) = \mathsf{false}, \\
\mathsf{eqNat}\ (\mathsf{succ}\ x) = g\, x, & g\ x\ \mathsf{zero} = \mathsf{false}, & g\ x\ (\mathsf{succ}\ y) = \mathsf{eqNat}\ x\ y
\end{array}$$

The last example is the inductive-recursive definition [8] of a universe containing a code of the type of natural numbers and the dependent product formation, defined in a mutual recursive way with its corresponding decoding function:

$$\begin{array}{rl}
\mathsf{rec}\ (\mathsf{V}, \mathsf{T})\ : & \Sigma\, X : \mathsf{U}\, .\, X \to \mathsf{U} \\
= & (\ \mathsf{Sum}\ (\mathsf{nat} \mid \mathsf{pi}\, (\Sigma\, x : \mathsf{V}\, .\, \mathsf{T}\, x \to \mathsf{V})), \\
 & \ \ \mathsf{fun}\ (\mathsf{nat} \to \mathsf{Nat} \mid \mathsf{pi}\ (x, f) \to \Pi\, y : \mathsf{T}\, x\, .\, \mathsf{T}\, (f\, y)))\ ;
\end{array}$$

6.4 Operational Semantics

In order to define the semantics of the language, it is necessary to first define the set of values (figure 6.2).

6.4.1 Values

A *value* represents an open expression in weak head normal form. It is either a *neutral value* $[k]$ which represents an expression whose computation stopped because of an attempt to compute a variable, or a *canonical value*, the form of which makes clear the head construction of an expression: λ-abstraction λf, Π-abstraction $\Pi\, t\, g$, etc.

The neutral value x_n is a primitive (not defined) constant which is used to represent the value of a free variable. It is a constant about which we know nothing. It is called a *generic value* in [6].

Other neutral values are built up from evaluation contexts in which neutral values are attempted to be computed. For instance, we obtain the neutral value $k\,v$ when trying to evaluate an application and the value of the function is the neutral value k and the argument is v. Similarly, the neutral values $k.1$ and $k.2$ are results from trying to project from a neutral value k. The neutral value $\langle S, \rho\rangle\ k$ is the result from trying to apply a choice function $\mathsf{fun}\,S$ to a neutral value k in an environment ρ. Neutral values are called *accumulators* by Grégoire and Leroy [10].

6.4.2 Value operations

There is a small set of functions defined on values. They are in general not defined for all arguments, e.g. the projections are not defined for functions. This does not lead to problems since the operations are only applied when evaluating well-typed expressions.

There is a function which instantiates a function closure to a value. It is defined by:

$$\begin{array}{lcl}
\mathsf{inst}\langle \lambda p.M, \rho\rangle\, v & = & [\![M]\!](\rho, p = v) \\
\mathsf{inst}(f \circ c)\, v & = & \mathsf{inst}\, f\,(c\, v)
\end{array}$$

Application $\mathsf{app}\,u\,v$ of values is defined using instantiation. Notice how a neutral value is built up in the case that the function is a neutral value:

$$\begin{array}{lcl}
\mathsf{app}(\lambda f)\, v & = & \mathsf{inst}\, f\, v \\
\mathsf{app}(\mathsf{fun}\langle S, \rho\rangle)(c_i\, v) & = & \mathsf{app}([\![M_i]\!]\rho)\, v \\
 & & \text{where } S = (c_1 \to M_1 \mid \cdots \mid c_n \to M_n) \\
\mathsf{app}(\mathsf{fun}\, s)[k] & = & [s\, k] \\
\mathsf{app}[k]\, v & = & [k\, v]
\end{array}$$

The projection function for pairs of values follows the same pattern:

$$\begin{array}{lcl}
(u, v).1 & = & u \\
[k].1 & = & [k.1] \\
(u, v).2 & = & v \\
[k].2 & = & [k.2]
\end{array}$$

The function to look up the value $\rho(x)$ of a variable x in an envi-

$$\begin{array}{lcl}
[\![\lambda\, p\,.\, M]\!]\rho & = & \langle \lambda p.M, \rho\rangle \\
[\![x]\!]\rho & = & \rho(x) \\
[\![M\, N]\!]\rho & = & \mathsf{app}([\![M]\!]\rho)([\![N]\!]\rho) \\
[\![\Pi\, p : A\,.\, B]\!]\rho & = & \Pi\,([\![A]\!]\rho)\,\langle \lambda p.B, \rho\rangle \\
[\![\mathsf{U}]\!]\rho & = & \mathsf{U} \\
[\![D; M]\!]\rho & = & [\![M]\!](\rho, D)
\end{array}$$

$$\begin{array}{lcl}
[\![M, N]\!]\rho & = & ([\![M]\!]\rho, [\![N]\!]\rho) \\
[\![0]\!]\rho & = & 0 \\
[\![M.1]\!]\rho & = & ([\![M]\!]\rho).1 \\
[\![M.2]\!]\rho & = & ([\![M]\!]\rho).2 \\
[\![\Sigma\, p : A\,.\, B]\!]\rho & = & \Sigma\,([\![A]\!]\rho)\,\langle \lambda p.B, \rho\rangle \\
[\![\mathbf{1}]\!]\rho & = & \mathbf{1}
\end{array}$$

$$\begin{array}{lcl}
[\![c\, M]\!]\rho & = & c\,([\![M]\!]\rho) \\
[\![\mathsf{fun}\, S]\!]\rho & = & \mathsf{fun}\langle S, \rho\rangle \\
[\![\mathsf{Sum}\, S]\!]\rho & = & \mathsf{Sum}\langle S, \rho\rangle
\end{array}$$

Fig. 6.3. Semantics of Mini-TT

ronment ρ is only defined for ρ in which x is defined. Type-checking guarantees that this is the case.
If x is in p,

$$\begin{array}{lcl}
(\rho, p = v)(x) & = & \mathsf{proj}^p_x(v) \\
(\rho, p : A = M)(x) & = & \mathsf{proj}^p_x([\![M]\!]\rho) \\
(\rho, \mathsf{rec}\, p : A = M)(x) & = & \mathsf{proj}^p_x([\![M]\!](\rho, \mathsf{rec}\, p : A = M))
\end{array}$$

If x is not in p,

$$\begin{array}{lcl}
(\rho, p = v)(x) & = & \rho(x) \\
(\rho, D)(x) & = & \rho(x)
\end{array}$$

The notation $\mathsf{proj}^p_x(v)$ is well-defined under the precondition that x is in p.

$$\begin{array}{lcll}
\mathsf{proj}^x_x(v) & = & v & \\
\mathsf{proj}^{(p_1,p_2)}_x(v) & = & \mathsf{proj}^{p_1}_x(v.1) & \text{if } x \text{ is in } p_1, \\
\mathsf{proj}^{(p_1,p_2)}_x(v) & = & \mathsf{proj}^{p_2}_x(v.2) & \text{if } x \text{ is in } p_2
\end{array}$$

6.4.3 Semantics

In figure 6.3 we give the semantics of Mini-TT by equations of the form $[\![M]\!]\rho = v$, meaning that the expression M evaluates to the value v in the environment ρ.

$$\begin{array}{lcl}
E & ::= & \lambda\, \mathsf{x}_i\, .\, E \mid \Pi\, \mathsf{x}_i : E_1\, .\, E_2 \mid \mathsf{U} \mid [K] \\
 & & E_1, E_2 \mid 0 \mid \Sigma\, \mathsf{x}_i : E_1\, .\, E_2 \mid \mathbf{1} \\
 & & c\, E \mid \mathsf{fun}\langle S, \alpha\rangle \mid \mathsf{Sum}\langle S, \alpha\rangle \\
K & ::= & \mathsf{x}_i \mid K E \mid K.1 \mid K.2 \mid \langle S, \alpha\rangle\, K \\
\alpha & ::= & () \mid (\alpha, p = E) \mid (\alpha, D)
\end{array}$$

Fig. 6.4. Normal expressions

$$\begin{array}{lcl}
\mathsf{R}_i(\lambda f) & = & \lambda\, \mathsf{x}_i\, .\, \mathsf{R}_{i+1}(\mathsf{inst}\, f[\mathsf{x}_i]) \\
\mathsf{R}_i(u, v) & = & (\mathsf{R}_i\, u, \mathsf{R}_i\, v) \\
\mathsf{R}_i\, 0 & = & 0 \\
\mathsf{R}_i(c\, v) & = & c\, (\mathsf{R}_i\, v) \\
\mathsf{R}_i(\mathsf{fun}\langle S, \rho\rangle) & = & \mathsf{fun}\langle S, \mathsf{R}_i\, \rho\rangle \\
\mathsf{R}_i(\mathsf{Sum}\langle S, \rho\rangle) & = & \mathsf{Sum}\langle S, \mathsf{R}_i\, \rho\rangle \\
\mathsf{R}_i\, \mathsf{U} & = & \mathsf{U} \\
\mathsf{R}_i\, \mathbf{1} & = & \mathbf{1} \\
\mathsf{R}_i(\Pi\, t\, g) & = & \Pi\, \mathsf{x}_i : \mathsf{R}_i\, t\, .\, \mathsf{R}_{i+1}(\mathsf{inst}\, g[\mathsf{x}_i]) \\
\mathsf{R}_i(\Sigma\, t\, g) & = & \Sigma\, \mathsf{x}_i : \mathsf{R}_i\, t\, .\, \mathsf{R}_{i+1}(\mathsf{inst}\, g[\mathsf{x}_i]) \\
\mathsf{R}_i[k] & = & [\mathsf{R}_i\, k] \\
 & & \\
\mathsf{R}_i\, \mathsf{x}_j & = & \mathsf{x}_j \\
\mathsf{R}_i(k\, v) & = & (\mathsf{R}_i\, k)(\mathsf{R}_i\, v) \\
\mathsf{R}_i(k.1) & = & (\mathsf{R}_i\, k).1 \\
\mathsf{R}_i(k.2) & = & (\mathsf{R}_i\, k).2 \\
\mathsf{R}_i(\langle S, \rho\rangle\, k) & = & \langle S, \mathsf{R}_i\, \rho\rangle\, (\mathsf{R}_i\, k) \\
 & & \\
\mathsf{R}_i(\rho, p = v) & = & \mathsf{R}_i\, \rho,\ p = \mathsf{R}_i\, v \\
\mathsf{R}_i(\rho, D) & = & \mathsf{R}_i\, \rho, D \\
\mathsf{R}_i() & = & ()
\end{array}$$

Fig. 6.5. The readback notation

6.5 Normal expressions and Readback

The readback function R_i takes a value to a normal expression (figure 6.2). The purpose of R_i is to facilitate convertibility checking. Notice that normal expressions are first-order objects, and have a decidable (syntactic) equality. Two convertible values are mapped to the same normal expression (i.e. identical, including choice of bound variables). This is similar to [10].

We overload the notation $\mathsf{R}_i(-)$ $(i \in \mathbb{N})$ for three cases: the readback of a value $\mathsf{R}_i\, v$ is a normal expression E, that of a neutral value $\mathsf{R}_i\, k$ is a neutral expression K, and that of an environment $\mathsf{R}_i\, \rho$ is a normal environment α.

6.6 Typing Rules

Typing context A typing context consists of an environment ρ and a *type environment* Γ:

$$\Gamma ::= () \mid \Gamma,\ x : t$$

The lookup operation $\Gamma(x)$ is expressed not as a function but as an inductive predicate since it may fail and signals incorrectness of expression being type checked.

$$\frac{}{(\Gamma, x : t)(x) \to t} \qquad \frac{\Gamma(x) \to t}{(\Gamma, y : t')(x) \to t}\ y \neq x$$

That Γ is updated by binding $p : t = v$ to Γ' is written $\Gamma \vdash p : t = v \Rightarrow \Gamma'$. It decomposes the pattern binding to bindings of simple variables while checking that the shape of p fits the type t. The bound value v is needed to compute the type of subpatterns of p.

$$\frac{}{\Gamma \vdash x : t = v \Rightarrow \Gamma, x : t} \qquad \frac{}{\Gamma \vdash \_ : t = v \Rightarrow \Gamma}$$

$$\frac{\Gamma \vdash p_1 : t_1 = v.1 \Rightarrow \Gamma_1 \quad \Gamma_1 \vdash p_2 : \mathsf{inst}\, g(v.1) = v.2 \Rightarrow \Gamma_2}{\Gamma \vdash (p_1, p_2) : \Sigma\, t_1\, g = v \Rightarrow \Gamma_2}$$

6.6.1 Overview

There are four forms of judgements.

checkD	$\rho, \Gamma \vdash_l D \Rightarrow \Gamma'$	D is a correct declaration and extends Γ to Γ'
checkT	$\rho, \Gamma \vdash_l A$	A is a correct type expression
check	$\rho, \Gamma \vdash_l M \Leftarrow t$	M is a correct expression of the given type t
checkI	$\rho, \Gamma \vdash_l M \Rightarrow t$	M is a correct expression and its type is inferred to be t

The inference rules are syntax directed and constitute a standard bidirectional type-checking (semi-)algorithm. It is an important property of the checking algorithm that $[\![M]\!]\rho$ is never computed without first checking that M is well-formed.

checkD: Check that a declaration is correct

$$\frac{\rho, \Gamma \vdash_l A \quad \rho, \Gamma \vdash_l M \Leftarrow t \quad \Gamma \vdash p : t = [\![M]\!]\rho \Rightarrow \Gamma_1}{\rho, \Gamma \vdash_l p : A = M \Rightarrow \Gamma_1} \; (t = [\![A]\!]\rho)$$

$$\frac{\begin{array}{c}\rho, \Gamma \vdash_l A \\ \Gamma \vdash p : t = [\mathsf{x}_l] \Rightarrow \Gamma_1 \\ (\rho, p = [\mathsf{x}_l]), \Gamma_1 \vdash_{l+1} M \Leftarrow t \\ \Gamma \vdash p : t = v \Rightarrow \Gamma_2\end{array}}{\rho, \Gamma \vdash_l \mathsf{rec}\, p : A = M \Rightarrow \Gamma_2} \; \left(\begin{array}{lcl} t & = & [\![A]\!]\rho, \\ v & = & [\![M]\!](\rho, \mathsf{rec}\, p : A = M)\end{array}\right)$$

The rule for a let binding is as expected. We check that A is a type, M is an expression of that type, and extend Γ while checking that p fits the type. In the rule for a letrec binding, the body M is checked in a temporarily extended context where p is bound to a generic value. This means that while checking M, recursively defined identifiers are treated as fresh constants about which we assume nothing but their typing. Once M is checked, Γ is extended using the 'real' value $[\![M]\!](\rho, \mathsf{rec}\, p : A = M)$ for p.

checkT: Check that something is a type

$$\frac{}{\rho, \Gamma \vdash_l \mathsf{U}} \qquad \frac{\rho, \Gamma \vdash_l A \quad \Gamma \vdash p : [\![A]\!]\rho = [\mathsf{x}_l] \Rightarrow \Gamma_1 \quad (\rho, p = [\mathsf{x}_l]), \Gamma_1 \vdash_{l+1} B}{\rho, \Gamma \vdash_l \Pi p : A \,.\, B}$$

$$\frac{\rho, \Gamma \vdash_l A \quad \Gamma \vdash p : [\![A]\!]\rho = [\mathsf{x}_l] \Rightarrow \Gamma_1 \quad (\rho, p = [\mathsf{x}_l]), \Gamma_1 \vdash_{l+1} B}{\rho, \Gamma \vdash_l \Sigma p : A \,.\, B}$$

$$\frac{\rho, \Gamma \vdash_l A \Leftarrow \mathsf{U}}{\rho, \Gamma \vdash_l A} \; \text{(if other rules are not applicable)}$$

If A is expected to be a type but not any of U, Π, or Σ, then it must be a small type of type U (the last rule).

check: Check that an expression has a given type

$$\frac{\Gamma \vdash p : t = [\mathsf{x}_l] \Rightarrow \Gamma_1 \quad (\rho, p = [\mathsf{x}_l]), \Gamma_1 \vdash_{l+1} M \Leftarrow \mathsf{inst}\, g\, [\mathsf{x}_l]}{\rho, \Gamma \vdash_l \lambda p \,.\, M \Leftarrow \Pi\, t\, g}$$

$$\frac{\rho, \Gamma \vdash_l M \Leftarrow t \quad \rho, \Gamma \vdash_l N \Leftarrow \mathsf{inst}\, g(\llbracket M \rrbracket \rho)}{\rho, \Gamma \vdash_l (M, N) \Leftarrow \Sigma\, t\, g}$$

$$\frac{\rho, \Gamma \vdash_l M \Leftarrow \llbracket A_i \rrbracket \nu}{\rho, \Gamma \vdash_l c_i\, M \Leftarrow \mathsf{Sum}\langle c_1\, A_1 \mid \cdots \mid c_n\, A_n,\ \nu\rangle}$$

$$\frac{\begin{array}{c}\rho, \Gamma \vdash_l M_1 \Leftarrow \Pi\, (\llbracket A_1 \rrbracket \nu)\, (g \circ c_1) \\ \cdots \\ \rho, \Gamma \vdash_l M_n \Leftarrow \Pi\, (\llbracket A_n \rrbracket \nu)\, (g \circ c_n)\end{array}}{\begin{array}{l}\rho, \Gamma \vdash_l \mathsf{fun}(c_1 \to M_1 \mid \cdots \mid c_n \to M_n) \Leftarrow \\ \qquad\qquad \Pi\, (\mathsf{Sum}\langle c_1 : A_1 \mid \cdots \mid c_n : A_n, \nu\rangle)\, g\end{array}}$$

$$\frac{\rho, \Gamma \vdash_l D \Rightarrow \Gamma_1 \quad (\rho, D), \Gamma_1 \vdash_l M \Leftarrow t}{\rho, \Gamma \vdash_l D; M \Leftarrow t}$$

$$\frac{}{\rho, \Gamma \vdash_l 0 \Leftarrow \mathbf{1}} \qquad \frac{}{\rho, \Gamma \vdash_l \mathbf{1} \Leftarrow \mathsf{U}}$$

$$\frac{\rho, \Gamma \vdash_l A \Leftarrow \mathsf{U} \quad \Gamma \vdash p : \llbracket A \rrbracket \rho = [\mathsf{x}_l] \Rightarrow \Gamma_1 \quad (\rho, p = [\mathsf{x}_l]), \Gamma_1 \vdash_{l+1} B \Leftarrow \mathsf{U}}{\rho, \Gamma \vdash_l \Pi p : A \,.\, B \Leftarrow \mathsf{U}}$$

$$\frac{\rho, \Gamma \vdash_l A \Leftarrow \mathsf{U} \quad \Gamma \vdash p : \llbracket A \rrbracket \rho = [\mathsf{x}_l] \Rightarrow \Gamma_1 \quad (\rho, p = [\mathsf{x}_l]), \Gamma_1 \vdash_{l+1} B \Leftarrow \mathsf{U}}{\rho, \Gamma \vdash_l \Sigma p : A \,.\, B \Leftarrow \mathsf{U}}$$

$$\frac{\rho, \Gamma \vdash_l A_1 \Leftarrow \mathsf{U} \quad \cdots \quad \rho, \Gamma \vdash_l A_n \Leftarrow \mathsf{U}}{\rho, \Gamma \vdash_l \mathsf{Sum}(c_1\, A_1 \mid \cdots \mid c_n\, A_n) \Leftarrow \mathsf{U}}$$

$$\frac{\rho, \Gamma \vdash_l M \Rightarrow t' \quad \mathsf{R}_l\, t = \mathsf{R}_l\, t'}{\rho, \Gamma \vdash_l M \Leftarrow t} \ \text{(if other rules are not applicable)}$$

This deals with expressions in canonical forms (weak head normal forms). For an expression in a non-canonical form, the last rule infers its type and checks that the inferred type is equal to the expected one. This is the single place where type checking uses conversion checking.

In the rule for a case-analysis function, it must be checked against a Π type whose domain is a Sum type. For simplicity, we require the constructors in case branches to match exactly the constructors listed in the Sum type, including the order. From the right hand side of the equation

$$\mathsf{app}([\![\mathsf{fun}(c_1 \to M_1 \mid \cdots \mid c_n \to M_n)]\!]\rho)(c_i\, v) = \mathsf{app}([\![M_i]\!]\rho)\, v$$

we expect the branch expression M_i to have a Π type with the domain $[\![A_i]\!]\nu$. The closure $g \circ c_i$ in the codomain part is what is needed to make both sides of the equation to have the same type, namely $\mathsf{inst}\, g(c_i\, v)$.

The rules for Π, Σ, $\mathbf{1}$, and Sum here make the universe U to be directly closed under those operations, unlike the type **Set** of Logical Framework.

checkI: Infer the type of an expression

$$\frac{\Gamma(x) \to t}{\rho, \Gamma \vdash_l x \Rightarrow t} \qquad \frac{\rho, \Gamma \vdash_l M \Rightarrow \Pi\, t\, g \quad \rho, \Gamma \vdash_l N \Leftarrow t}{\rho, \Gamma \vdash_l M\, N \Rightarrow \mathsf{inst}\, g([\![N]\!]\rho)}$$

$$\frac{\rho, \Gamma \vdash_l M \Rightarrow \Sigma\, t\, g}{\rho, \Gamma \vdash_l M.1 \Rightarrow t} \qquad \frac{\rho, \Gamma \vdash_l M \Rightarrow \Sigma\, t\, g}{\rho, \Gamma \vdash_l M.2 \Rightarrow \mathsf{inst}\, g(([\![M]\!]\rho).1)}$$

We check and infer types of expressions in non-canonical forms here.

6.7 Metamathematical remarks

As we explained in the introduction, the work [4, 7] should provide a general *semantical* condition ensuring termination of type-checking: it is enough that the *strict* denotational semantics of the program is $\neq\perp$. As in [4, 7], one can ensure this by proving *totality* of the program. In turn, there are sufficient purely syntactical criterion ensuring totality. One such criteria is for instance *size-change termination* [11, 15].

6.8 Variations

NBE and η-conversion We can adopt the typed NBE algorithm by Abel, Dybjer, and Coquand [2] for our evaluation to obtain the version of Mini-TT with η-conversion. There are two points to modify our presentation of Mini-TT. First, when type checking under a binder, we extend a context not by a generic value $[\mathsf{x}_l]$ $(l = |\rho|)$ but by its reflected form $\uparrow^t [\mathsf{x}_l]$ where t is the type of the generic value. Second, when we compare the expected type t and the inferred type t' in the last of **check**, we compare the readbacks of their reified form $\mathsf{R}_i \Downarrow t$ and $\mathsf{R}_i \Downarrow t'$. These modifications make the comparison to be between η-long normal forms, thus making Mini-TT a language type checked with η-conversion.

Higher order values A function closures f is a first order representation of a semantic function from values to values. We do not need to "look inside" it (cf. [10]). This can be made clear by replacing closures with these semantic functions themselves, thus making values higher order. Then, closure instantiation and constructions are replaced by the following.

$$\begin{array}{lcl} \mathsf{inst}\, f\, v & = & f\, v \\ \langle \lambda p.M, \rho \rangle & = & (v \mapsto [\![M]\!](\rho, p = v)) \\ f \circ c & = & (v \mapsto f(c\, v)) \end{array}$$

6.9 Conclusion

We have presented a dependently typed language Mini-TT with its semantics and type checking rules. Mini-TT has dependent products, dependent sums and unit type, labelled sums, recursive definitions, and pattern abstractions and bindings.

Mini-TT is a step towards a simple and definitive core language for the proof-assistant Agda [3] based on versions of Martin-Löf Type Theory. To make development of large proofs and programs feasible, the full language must support various advanced features such as incomplete terms with meta-variables and synthesis of implicit arguments. Directly giving semantics to them and justifying its complex implementation is difficult. Our approach is to translate the full language to a well-understood simple core language. We would have a simple theory and implementation of the core language, with respect to which a full-fledged proof assistant is specified, implemented, and tested.

Our future work is towards that goal. This includes a strong normalization theorem for Mini-TT using the denotational semantics of [4, 7], non-uniform inductive families of types, universe hierarchy, proven correct compilation to abstract machine code as in [10], etc.

Bibliography

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990.

[2] Andreas Abel, Klaus Aehlig, and Peter Dybjer. Normalization by evaluation for Martin-Löf type theory with one universe. *Electronic Notes in Theoretical Computer Science*, 173:17–39, 2007.

[3] Agda homepage. http://unit.aist.go.jp/cvs/Agda/.

[4] Ulrich Berger. Strong normalization for applied lambda calculi. *Logical Methods in Computer Science*, 1(2), 2005.

[5] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *LISP and Functional Programming*, pages 13–27, 1986.

[6] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1–3):167–177, 1996.

[7] Thierry Coquand and Arnaud Spiwack. A proof of strong normalisation using domain theory. In *LICS*, pages 307–316. IEEE Computer Society, 2006.

[8] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, 2000.

[9] B. Grégoire. *Compilation des termes de preuves: un (nouveau) mariage entre Coq et Ocaml.* Thése de doctorat, spécialité informatique, Université Paris 7, École Polytechnique, France, December 2003.

[10] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246. ACM Press, 2002.

[11] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Conference Record of the Twenty-eighth Annual ACM Symposium on Principles of Programming Languages*, volume 28 of *ACM SIGPLAN Notices*, pages 81–92. ACM press, january 2001.

[12] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *Monographs on Computer Science*. Oxford University Press, 1990.

[13] Bengt Nordström, Kent Petersson, and Jan M. Smith. Martin-Löf's type theory. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 5. Oxford Science Publications, 2000.

[14] Michael Pellauer, Markus Forsberg, and Aarne Ranta. BNF Converter multilingual front-end generation from labelled BNF grammars. Technical Report 2004-09, Department of Computing Science, Chalmers University of Technology and Göteborg University, 2004. available from http://www.cs.chalmers.se/˜markus/BNFC/.

[15] David Wahlstedt. *Dependent Type Theory with Parameterized First-Order Data Types and Well-Founded Recursion.* PhD thesis, Chalmers University of Technology, 2007. ISBN 978-91-7291-979-2.

Appendix: Implementation

The inference rules are directly translated to Haskell using a simple error monad `G` a. The Haskell typing of the routines that corresponds to the four forms of judgements are:

```
data G a = Success a | Fail Name

instance  Monad G  where
    (Success x) >>= k     =  k x
    Fail s   >>= k        =  Fail s
    return                =  Success
    fail                  =  Fail

checkD :: Int -> Rho -> Gamma -> Decl -> G Gamma
checkT :: Int -> Rho -> Gamma -> Exp -> G ()
check  :: Int -> Rho -> Gamma -> Exp -> TVal -> G ()
checkI :: Int -> Rho -> Gamma -> Exp -> G TVal
```

If these routines return without producing error messages, then there are derivations that conclude corresponding judgements. The clause for the application rule of checkI judgement is

$$\frac{\rho, \Gamma \vdash_l M \Rightarrow \Pi\, t\, g \quad \rho, \Gamma \vdash_l N \Leftarrow t}{\rho, \Gamma \vdash_l M\, N \Rightarrow \mathsf{inst}\, g([\![N]\!]\rho)}$$

```
checkI k rho gma (EApp e1 e2) =
  do t1 <- checkI k rho gma e1
     (t, g) <- extPiG t1
     check k rho gma e2 t
     return (g * eval e2 rho)
  where
  extPiG (Pi t g) = return (t, g)
  extPiG u        = fail ("extPiG " ++ showVal u)
```

The implementation supposes a parser function. One can either write directly a parser in Haskell, or use the *BNF Converter* compiler construction tool [14]. From a description of concrete syntax in a labelled BNF grammar, BNFC generates modules for the data type for abstract syntax trees, a parser, and a pretty printer.

The implementation can be obtained from `http://www.cs.chalmers.se/Cs/Research/Logic/Mini-TT/`

```
-----------------------------------------
-- Main module
-----------------------------------------

module Main where
```

```
import Prelude hiding ((*))

-----------------------------------------------------------
-- Expressions
------------------------------------------------------------

type Name = String

data Exp =
   ELam Patt Exp
 | ESet
 | EPi Patt Exp Exp
 | ESig Patt Exp Exp
 | EOne
 | Eunit
 | EPair Exp Exp
 | ECon Name Exp
 | ESum Branch
 | EFun Branch
 | EFst Exp
 | ESnd Exp
 | EApp Exp Exp
 | EVar Name
 | EVoid
 | EDec Decl Exp
  deriving (Eq,Ord,Show)

data Decl =
   Def Patt Exp Exp
 | Drec Patt Exp Exp
  deriving (Eq,Ord,Show)

data Patt =
   PPair Patt Patt
 | Punit
 | PVar Name
  deriving (Eq,Ord,Show)

type Branch = [(Name,Exp)]
```

```
------------------------------------------------------------
-- Values
------------------------------------------------------------

data Val =
     Lam Clos
  |  Pair Val Val
  |  Con Name Val
  |  Unit
  |  Set
  |  Pi  Val Clos
  |  Sig Val Clos
  |  One
  |  Fun SClos
  |  Sum SClos
  |  Nt Neut
  deriving Show

data Neut = Gen  Int
          | App  Neut Val
          | Fst  Neut
          | Snd  Neut
          | NtFun SClos Neut
  deriving Show

type SClos = (Branch, Rho)

-- Function closures
data Clos = Cl Patt Exp Rho | ClCmp Clos Name
  deriving Show

-- instantiation of a closure by a value
(*) :: Clos -> Val -> Val
(Cl p e rho) * v = eval e (UpVar rho p v)
(ClCmp f c ) * v = f * Con c v

mkCl :: Patt -> Exp -> Rho -> Clos
mkCl p e rho = Cl p e rho
```

```
clCmp :: Clos -> Name -> Clos
clCmp g c  = ClCmp g c

get s [] = error ("get " ++ show s)
get s ((s1,u):us) | s == s1 = u
get s ((s1,u):us)           = get s us

app :: Val -> Val -> Val
app (Lam f)             v        = f * v
app (Fun (ces, rho)) (Con c v)   =
  app (eval (get c es) rho) v
app (Fun s)        (Nt k)        = Nt(NtFun s k)
app (Nt k)             m         = Nt(App k m)
app w u = error "app "

vfst :: Val -> Val
vfst (Pair u1 _) = u1
vfst (Nt k)      = Nt(Fst k)
vfst w = error "vfst "

vsnd :: Val -> Val
vsnd (Pair _ u2) = u2
vsnd (Nt k)      = Nt(Snd k)
vsnd w =  error "vsnd "

---------------------------------------------
-- Environment
---------------------------------------------

data Rho = RNil | UpVar Rho Patt Val | UpDec Rho Decl
  deriving Show

getRho :: Rho -> Name -> Val
getRho (UpVar rho p v) x | x `inPat` p = patProj p x v
                         | otherwise   = getRho rho x
getRho (UpDec rho (Def  p _ e)) x
  | x `inPat` p = patProj p x (eval e rho)
  | otherwise   = getRho rho x
getRho rho0@(UpDec rho (Drec p _ e)) x
  | x `inPat` p = patProj p x (eval e rho0)
  | otherwise   = getRho rho x
getRho RNil _ = error "getRho"

inPat :: Name -> Patt -> Bool
inPat x (PVar y)      = x == y
inPat x (PPair p1 p2) = inPat x p1 || inPat x p2
inPat _ Punit         = False

patProj :: Patt -> Name -> Val -> Val
patProj (PVar y)      x v | x == y       = v
patProj (PPair p1 p2) x v | x `inPat` p1 = patProj p1 x (vfst v)
                          | x `inPat` p2 = patProj p2 x (vsnd v)
patProj _ _ _ = error "patProj"

lRho :: Rho -> Int
lRho RNil           = 0
lRho (UpVar rho _ _) = lRho rho + 1
lRho (UpDec rho _  ) = lRho rho

eval :: Exp -> Rho -> Val
eval e0 rho = case e0 of
    ESet          -> Set
    EDec d e      -> eval e (UpDec rho d)
    ELam p e      -> Lam $ mkCl p e rho
    EPi  p a b    -> Pi  (eval a rho) $ mkCl p b rho
    ESig p a b    -> Sig (eval a rho) $ mkCl p b rho
    EOne          -> One
    Eunit         -> Unit
    EFst e        -> vfst (eval e rho)
    ESnd e        -> vsnd (eval e rho)
    EApp e1 e2    -> app (eval e1 rho) (eval e2 rho)
    EVar x        -> getRho rho x
    EPair e1 e2   -> Pair  (eval e1 rho) (eval e2 rho)
    ECon c e1     -> Con c (eval e1 rho)
    ESum cas     -> Sum (cas, rho)
    EFun ces     -> Fun (ces, rho)
    e -> error $ "eval: " ++ show e

------------------------------------------------------------
-- Normal forms
```

```
-----------------------------------------------------------

data NExp =
     NLam Int NExp
  |  NPair NExp NExp
  |  NCon Name NExp
  |  NUnit
  |  NSet
  |  NPi NExp Int NExp
  |  NSig NExp Int NExp
  |  NOne
  |  NFun NSClos
  |  NSum NSClos
  |  NNt NNeut
  deriving (Eq,Show)

data NNeut = NGen Int
          | NApp NNeut NExp
          | NFst NNeut
          | NSnd  NNeut
          | NNtFun NSClos NNeut
  deriving (Eq,Show)

type NSClos = (Branch, NRho)

data NRho = NRNil | NUpVar NRho Patt NExp | NUpDec NRho Decl
  deriving (Eq,Show)

-------------------------------------------
-- Readback functions
-------------------------------------------

rbV :: Int -> Val  -> NExp

rbV k v0 = case v0 of
      Lam f        -> NLam k (rbV (k+1) (f * genV k))
      Pair u v     -> NPair (rbV k u) (rbV k v)
      Con  c v     -> NCon  c (rbV k v)
      Unit         -> NUnit
      Set          -> NSet
      Pi  t g       -> NPi (rbV k t) k (rbV (k+1) (g * genV k))
      Sig t g       -> NSig (rbV k t) k (rbV (k+1) (g * genV k))
      One           -> NOne
      Fun (s,rho)   -> NFun (s,rbRho k rho)
      Sum (s,rho) -> NSum (s,rbRho k rho)
      Nt l          -> NNt (rbN k l)

rbN :: Int -> Neut -> NNeut
rbN i k0 = case k0 of
      Gen j   -> NGen j
      App k m -> NApp (rbN i k) (rbV i m)
      Fst k   -> NFst (rbN i k)
      Snd k   -> NSnd (rbN i k)
      NtFun (s,rho) k -> NNtFun (s,rbRho i rho) (rbN i k)

rbRho :: Int -> Rho -> NRho
rbRho _ RNil = NRNil
rbRho i (UpVar rho p v) = NUpVar (rbRho i rho) p (rbV i v)
rbRho i (UpDec rho d  ) = NUpDec (rbRho i rho) d

------------------------------------------------
-- Error monad and type environment
------------------------------------------------

data G a = Success a | Fail Name

instance  Monad G  where
    (Success x) >>= k     =  k x
    Fail s   >>= k        =  Fail s
    return                =  Success
    fail                  =  Fail

type Gamma = [(Name, Val)]

lookupG :: (Show a, Eq a) => a -> [(a,b)] -> G b
lookupG s [] = fail ("lookupG " ++ show s)-- should never occur
lookupG s ((s1,u):us) | s == s1 = return u
lookupG s ((s1,u):us)           = lookupG s us

-- Updating type environment   Gamma |- p : t = u => Gamma'
```

```
upG :: Gamma -> Patt -> Val -> Val -> G Gamma
upG gma Punit         _         _ = return gma
upG gma (PVar x)      t         _ = return $ (x,t):gma
upG gma (PPair p1 p2) (Sig t g) v =
  do gma1 <- upG gma p1 t (vfst v)
     upG gma1 p2 (g * vfst v) (vsnd v)
upG _   p             _         _ =
  fail $ "upG: p = " ++  show p

-------------------------------------------------
-- Type checking rules
-------------------------------------------------

genV :: Int -> Val
genV k = Nt (Gen k)

checkT :: Int -> Rho -> Gamma -> Exp  -> G ()
check  :: Int -> Rho -> Gamma -> Exp  -> Val -> G ()
checkI :: Int -> Rho -> Gamma -> Exp  -> G Val
checkD :: Int -> Rho -> Gamma -> Decl -> G Gamma

checkT k rho gma e0 =
  case e0 of
    EPi  p a b -> do checkT k rho gma a
                     gma1 <- upG gma p (eval a rho) (genV k)
                     checkT (k+1) (UpVar rho p (genV k)) gma1 b
    ESig p a b -> checkT k rho gma (EPi p a b)
    ESet       -> return ()
    a          -> check k rho gma a Set

check k rho gma e0 t0 =
  case (e0, t0) of
    (ELam p e   , Pi  t g )->
       do let gen = genV k
          gma1 <- upG gma p t gen
          check (k+1) (UpVar rho p gen) gma1 e (g * gen)
    (EPair e1 e2, Sig t g )->
       do check k rho gma e1 t
          check k rho gma e2 (g * eval e1 rho)
    (ECon c e   , Sum (cas,rho1))->
```

```
      do a <- lookupG c cas
         check k rho gma e (eval a rho1)
    (EFun ces, Pi (Sum (cas, rho1)) g) ->
      if map fst ces == map fst cas
         then sequence_ [check k rho gma e (Pi (eval a rho1) (clCmp g c))
                        | ((c,e), (_,a)) <- zip ces cas]
        else fail "case branches does not match the data type"
    (Eunit      , One)-> return ()
    (EOne       , Set)-> return ()
    (EPi  p a b , Set)->
        do check k rho gma a Set
           let gen = genV k
           gma1 <- upG gma p (eval a rho) gen
           check (k+1) (UpVar rho p gen) gma1 b Set
    (ESig p a b , Set)-> check k rho gma (EPi p a b) Set
    (ESum cas, Set)  ->
        sequence_ [check k rho gma a Set | (_,a) <- cas]
    (EDec d e   , t  )-> do gma1 <- checkD k rho gma d
                            check k (UpDec rho d) gma1 e t
    (e          , t  )-> do t1 <- checkI k rho gma e
                            eqNf k t t1
  where
  eqNf :: Int -> Val -> Val -> G ()
  eqNf i m1 m2
    | e1 == e2  = return ()
    | otherwise = fail $ "eqNf: " ++ show e1 ++ "=/=" ++ show e2
    where e1 = rbV i m1
          e2 = rbV i m2

checkI k rho gma e0 =
  case e0 of
    EVar x     -> lookupG x gma
    EApp e1 e2 -> do t1 <- checkI k rho gma e1
                     (t, g) <- extPiG t1
                     check k rho gma e2 t
                     return (g * eval e2 rho)
    EFst e     -> do t <- checkI k rho gma e
                     (a,_) <- extSigG t
                     return a
    ESnd e     -> do t <- checkI k rho gma e
```

```
                     (_, g) <- extSigG t
                     return (g * vfst (eval e rho))

    e          -> fail ("checkI: " ++ show e)
  where
  extPiG :: Val -> G (Val, Clos)
  extPiG (Pi t g) = return (t, g)
  extPiG u        = fail ("extPiG " ++ showVal u)

  extSigG :: Val -> G (Val, Clos)
  extSigG (Sig t g) = return (t, g)
  extSigG u         = fail ("extSigG " ++ showVal u)

showVal u = show (rbV 0 u)

checkD k rho gma d@(Def  p a e) = do
  checkT k rho gma a
  let t = eval a rho
  check k rho gma e t
  upG gma p t (eval e rho)

checkD k rho gma d@(Drec p a e) = do
  checkT k rho gma a
  let t   = eval a rho
      gen = genV k
  gma1 <- upG gma p t gen
  check (k+1) (UpVar rho p gen) gma1 e t
  let v = eval e (UpDec rho d)
  upG gma p t v


-------------------------------------------------------
-- Main checking routines
-------------------------------------------------------

-- The input is checked as an expression of type One.
checkMain :: Exp -> G ()
checkMain e = check 0 RNil [] e One

-- checking a string input
```

```
checkStr :: String -> IO()
checkStr s =
  case parseExp $ myLex s of -- parsing using routines
    Fail msg -> putStrLn $ "Parse error: " ++ msg
    Success  (e,_)  ->
      case checkMain e of
        Fail  msg' ->
         putStrLn ("type-checking failed:\n" ++ msg')
        Success _  ->
         putStrLn ("type-checking succeded.")

-- checking the content of a file.
checkFile :: String -> IO()
checkFile file = checkStr =<< readFile file
```