An Algorithm for Type-Checking Dependent Types

Thierry Coquand

Chalmers University, S 41296, Göteborg, Sweden

Abstract

We present a simple type-checker for a language with dependent types and let expressions, with a simple proof of correctness.

Introduction

Type Theory provides an interesting approach to the problem of (interactive) proof-checking. Instead of introducing, like in LCF [10], an abstract data type of theorems, it uses the proofs-as-programs analogy and reduces the problem of proof checking to the problem of type-checking in a programming language with dependent types [5]. This approach presents several advantages, well described in [11,5], among those being the possibility of independent proof verification and of a uniform treatment for naming constants and theorems. It is crucial however for this approach to proof-checking to have a simple and reliable type-checking algorithm. Since the core part of such languages, like the ones described in [5,7], seems very simple, there may be some hope for such a short and simple type-checker for dependent types. Indeed, de Bruijn sketches such an algorithm in [5]. However, this last paper leaves unspecified the treatment of conversion of terms, and more importantly, the treatment of α-conversion, and names of variables.

Though this problem of α-conversion, and the related problem of the definition of substitution, may seem at first of small importance, there are both theoretical and practical evidence that it is an important issue for languages based on λ-calculus. We can cite here Abelson and Sussman [2]: "Despite the fact that substitution is a "straightforward idea", it turns out to be surprisingly complicated to give a rigorous mathematical definition of the substitution process ... Indeed, there is a long history of erroneous definitions of *substitution* in the literature of logic and programming semantics."

From a theoretical side, the problem of substitution and α-conversion are analysed in detail in A. Stoughton's paper [17], and one motivation behind the calculus of explicit substitution [1] was to handle precisely these problems. One other attempt for making precise the substitution operation is the substitution calculus of P. Martin-Löf, presented in the references [19]. Unexpectedly, R. Pollack discovered that this calculus is not closed under α-conversion [16], and this illustrates well the subtlety of this topic.

From the implementation side, it is known that the first implementation of substitution in Automath [5] was incorrect, and that most of the bugs in the implementation of LCF came from clashes of bound variables in strange situations [15]. How to handle names properly is seen as one of the main problem in the implementation of a language based on Type Theory by Hanna and Daeche [11].

Despite its importance, the problem of α-conversion is relatively seldom emphasised and analysed in the literature of type-checking dependent types. Few papers are explicit on this point, and even less try to argue about the correctness of their treatment of names of variables (for some exceptions, see [11,14,16,1,18]; the references [1,16,18] are more explicit about correctness issues). When the problem is analysed in detail, like in the proof of the "substitution lemma" in Stoy's book on denotational semantics [18], the arguments lack of conceptual content and it is difficult to grasp intuitively what makes the whole proof work.

The goal of this note is to present a simple type-checking algorithm for dependent types, with a simple proof of correctness. The main ingredient, which is the explicit introduction of *closures*, has been already suggested in [9], for the analysis of environment machines. In this way, we are completely explicit relatively to α-conversion, but we don't require complicated syntactical lemmata such as the substitution lemma.

For reason of simplicity, we have chosen to illustrate this method on the simplest possible type system with dependent types, namely a type system with dependent product and as only primitive type a type Type of all types. We prove only the soundness of our type-checking algorithm here. Indeed, it is known [8], that there is no decision procedure for the typing problem if we have a type of all types. However, with only minor variations, the same algorithm can be used as the basis of a decision procedure for Martin-Löf type theory or the normalising type systems described in [4]. We give in an appendix a Gofer/Haskell implementation [3].

1 Language and Semantics of Dependent Types

1.1 Expressions

Our expression language will be the one of a type theory with dependent type and a type of all types. Let Ide be an infinite set of identifiers. The set Exp of expressions is inductively defined by:

- $\mathsf{Ide} \subseteq \mathsf{Exp}$,
- $\mathsf{Type} \in \mathsf{Exp}$,
- if $M, N \in \mathsf{Exp},\ x \in \mathsf{Ide}$ then $\lambda x\ M,\ M\ N,\ (x : M)N \in \mathsf{Exp}$.

In the following, let A, B, C, M, N be expressions, x, y, z be identifiers.

For instance $(A : \mathsf{Type})(x : A)A$ will be the type of the polymorphic identity function $\mathsf{id} = \lambda A \lambda x\ x$. If B is a given type, $\mathsf{id}\ B$ is the identity function over the type B.

By structural induction, we can associate with any expression M its set of free variables $\mathsf{FV}(M)$ as usual.

1.2 Models

We don't need to make completely precise the notion of models, but only to list some general operations and properties that any "reasonable" model should have. We use the notion of models described in [13], due to Hindley and Longo. This notion can be traced back to Henkin in the framework of simply typed λ-calculus [12].

First, we define the set Env of environments associated with a given set D and $\mathsf{dom}(\rho) \subseteq \mathsf{Ide}$, defined for $\rho \in \mathsf{Env}$. The set Env consists of the empty environment $()$ and of the update environment $(\rho, x = d)$ for $\rho \in \mathsf{Env}$ and $x \in \mathsf{Ide},\ d \in \mathsf{D}$. We take furthermore $\mathsf{dom}(()) = \emptyset$ and $\mathsf{dom}((\rho, x = u)) = \mathsf{dom}(\rho) \cup \{x\}$. We define $\mathsf{lookup}\ x\ \rho$ for $x \in \mathsf{dom}(\rho)$, by taking $\mathsf{lookup}\ x\ (\rho, y = d)$ to be d if $x = y$ and $\mathsf{lookup}\ x\ \rho$ otherwise.

A model is a tuple $(\mathsf{D}, \mathsf{App}, \mathsf{eval}, \mathsf{Type}, :)$ where D is a set, $\mathsf{App} : \mathsf{D}{\rightarrow}[\mathsf{D}{\rightarrow}\mathsf{D}]$ a binary operation, and $\mathsf{eval}\ M\ \rho$ is an element of D for M expression and $\rho \in \mathsf{Env}$ such that $\mathsf{FV}(M) \subseteq \mathsf{dom}(\rho)$ [1]. The meaning of Type and the relation

[1] With respect to the presentation of Hindley-Seldin [13], we have made the following change: we replace environments as functions in $\mathsf{Ide}{\rightarrow}\mathsf{D}$, by suitable finite representations of these functions.

$:\subseteq \mathsf{D} \times \mathsf{D}$ are explained below.

First, we require that $(\mathsf{D}, \mathsf{App}, \mathsf{eval})$ forms a model of the untyped λ-calculus, that is:

- $\mathsf{App}\ (\mathsf{eval}\ (\lambda x\ M)\ \rho)\ a = \mathsf{eval}\ M\ (\rho, x = a)$,
- $\mathsf{eval}\ x\ \rho = \mathsf{lookup}\ x\ \rho$,
- $\mathsf{eval}\ (M_1\ M_2)\ \rho = \mathsf{App}\ (\mathsf{eval}\ M_1\ \rho)\ (\mathsf{eval}\ M_2\ \rho)$,
- if, for all elements $d \in \mathsf{D}$, we have $\mathsf{eval}\ M\ (\rho, x = d) = \mathsf{eval}\ N\ (\nu, y = d)$ then we have also $\mathsf{eval}\ (\lambda x\ M)\ \rho = \mathsf{eval}\ (\lambda y\ N)\ \nu$.

The first condition can be seen as a semantical version of β-conversion. The last condition is called Berry's condition in [13]. This is a quite elegant definition of model of λ-calculus, which is presented as exercise 11.9 in reference [13]. We write $[\![M]\!]$ for $\mathsf{eval}\ M\ ()$.

The situation is richer here because we have a special constant Type and a product operation. We add the conditions

- $\mathsf{eval}\ \mathsf{Type}\ \rho = \mathsf{Type}$ for all ρ,
- if $\mathsf{eval}\ A\ \rho = \mathsf{eval}\ C\ \nu$ and $\mathsf{eval}\ B\ (\rho, x = d) = \mathsf{eval}\ D\ (\nu, y = d)$, for all $d \in \mathsf{D}$, then we have also $\mathsf{eval}\ ((x : A)B)\ \rho = \mathsf{eval}\ ((y : C)D)\ \nu$.

Finally, we have to express in some way that some expressions denote *types*, that represent collections of objects. For this, we introduce a typing relation $:\subseteq \mathsf{D} \times \mathsf{D}$, written infix; the relation $a : d$ means intuitively that the value $a \in \mathsf{D}$ is of type d.

- $\mathsf{Type} : \mathsf{Type}$,
- $\mathsf{eval}\ ((x : A)B)\ \rho : \mathsf{Type}$ if first, $\mathsf{eval}\ A\ \rho : \mathsf{Type}$ and second, $a : \mathsf{eval}\ A\ \rho$ implies $\mathsf{eval}\ B\ (\rho, x = a) : \mathsf{Type}$,
- $\mathsf{App}\ c\ a : \mathsf{eval}\ B\ (\rho, x = a)$ if $c : \mathsf{eval}\ ((x : A)B)\ \rho$ and $a : \mathsf{eval}\ A\ \rho$,
- $\mathsf{eval}\ (\lambda y\ N)\ \nu : \mathsf{eval}\ ((x : A)B)\ \rho$ if $a : \mathsf{eval}\ A\ \rho$ implies $\mathsf{eval}\ N\ (\nu, y = a) : \mathsf{eval}\ B\ (\rho, x = a)$.

The following lemma can be proved by structural induction on the expression M.

Lemma 1 *If* $\mathsf{FV}(M) \subseteq \mathsf{dom}(\rho)$, $\mathsf{FV}(M) \subseteq \mathsf{dom}(\nu)$, *and* $\mathsf{lookup}\ x\ \rho = \mathsf{lookup}\ x\ \nu$ *for all* $x \in \mathsf{FV}(M)$, *then* $\mathsf{eval}\ M\ \rho = \mathsf{eval}\ M\ \nu$.

When later on we refer to a "model D", we shall mean by this any structure $(\mathsf{D}, \mathsf{App}, \mathsf{eval}, \mathsf{Type}, :)$ that satisfies all the conditions listed above.

2 The Type System

2.1 Expressions and Values

Let G be an infinite set of new variables, the *generic values*. We write $v_1, v_2, v_3, \ldots$ for elements of G.

Then we define by simultaneous induction the set of values V and the set of environments Env together with $\mathsf{dom}(\rho) \subseteq \mathsf{V}$ for $\rho \in \mathsf{Env}$:

- $\mathsf{G} \subseteq \mathsf{V}$,
- if $u, w \in \mathsf{V}$, then $u\ w \in \mathsf{V}$,
- $\mathsf{Type} \in \mathsf{V}$,
- if M is an expression, $\rho \in \mathsf{Env}$, $\mathsf{FV}(M) \subseteq \mathsf{dom}(\rho)$, then $M\rho \in \mathsf{V}$,
- $() \in \mathsf{Env}$, and $\mathsf{dom}(()) = \emptyset$,
- if $\rho \in \mathsf{Env}$, $x \in \mathsf{Ide}$, $u \in \mathsf{V}$, then $(\rho, x = u) \in \mathsf{Env}$, and $\mathsf{dom}((\rho, x = u)) = \mathsf{dom}(\rho) \cup \{x\}$.

We define $\mathsf{lookup}\ x\ \rho$ as before. In the following, assume $u, v, w \in \mathsf{V}$.

Any assignment $f \in \mathsf{G} \to \mathsf{D}$ extends uniquely to an $f \in \mathsf{V} \to \mathsf{D}$ such that

- $f\ \mathsf{Type} = \mathsf{Type}$,
- $f\ (u_1\ u_2) = \mathsf{App}\ (f\ u_1)\ (f\ u_2)$,
- $f\ (M\rho) = \mathsf{eval}\ M\ (f^*\rho)$,

where $f^*() = ()$ and $f^*(\rho, x = u) = ((f^*\rho), x = f\ u)$.

We can define inductively when a generic value occurs in a given value, and prove that $f\ u = g\ u$ if f and g agree on all generic values that occur in u.

2.2 Conversion Relation

Conversion applies only to *values*. Like for λ-calculus, we use the ordinary equality symbol $u = v$ to denote the fact that the values u and v are convertible. This relation is defined inductively as the least congruence such that (notice that this congruence appears only positively in the clauses that follow, and this is why we can define conversion inductively):

- $(\lambda x\ M)\rho\ v = M(\rho, x = v)$,
- $x(\rho, x = u) = u$,
- $x(\rho, y = v) = x\rho$, if $x \neq y$,
- $(M\ N)\rho = (M\rho)\ (N\rho)$,

– if v_k does not occur in ρ, ν, and $M(\rho, x = v_k) = N(\nu, y = v_k)$, then $(\lambda x\ M)\rho = (\lambda y\ N)\nu$,
– $\mathsf{Type}\ \rho = \mathsf{Type}$,
– if v_k does not occur in ρ, ν, and $A\rho = B\nu$, $B(\rho, x = v_k) = D(\nu, y = v_k)$, then $((x : A)B)\rho = ((y : C)D)\nu$.

The following Lemma is similar to Lemma 1, and follows from the fact that the set G of generic values is infinite.

Lemma 2 *If* $\mathsf{FV}(M) \subseteq \mathsf{dom}(\rho)$, $\mathsf{FV}(M) \subseteq \mathsf{dom}(\nu)$, *and* $\mathsf{lookup}\ x\ \rho = \mathsf{lookup}\ x\ \nu$ *for all* $x \in \mathsf{FV}(M)$, *then* $M\rho = M\nu$.

The following property expresses the soundness of our notion of conversion between values.

Proposition 1 *If* u_1 *and* u_2 *are convertible value, then for any model* D, *and any assignment* $f \in \mathsf{G} \to \mathsf{D}$, *we have* $f\ u_1 = f\ u_2$ *in* D.

Proof: We present the rule corresponding to Berry's condition, because this is the only delicate case.

We have to show $\mathsf{eval}\ (\lambda x\ M)\ (f^*\rho) = \mathsf{eval}\ (\lambda y\ N)\ (f^*\ \nu)$ in D, given that $M(\rho, x = v_k) = N(\nu, y = v_k)$ where v_k does not occur in ρ, ν. Let d be an arbitrary element of D. Let then g be the assignment that differs from f only on the variable v_k and such that $g\ v_k = d$. Because v_k does not occur in ρ, ν we have $f^*\rho = g^*\rho$ and $f^*\nu = g^*\nu$, and hence $((f^*\ \rho), x = d) = g^*\ (\rho, x = v_k)$ and $((f^*\ \nu), x = d) = g^*\ (\nu, x = v_k)$. So, by induction, $\mathsf{eval}\ M\ ((f^*\ \rho), x = d) = \mathsf{eval}\ N\ ((f^*\ \nu), y = d)$. Since this holds for all $d \in \mathsf{D}$, we get $\mathsf{eval}\ (\lambda x\ M)\ (f^*\rho) = \mathsf{eval}\ (\lambda y\ N)\ (f^*\ \nu)$ because Berry's condition holds for D. $\square$

2.3 Conversion algorithm

2.3.1 Weak Head Normal Form

The conversion algorithm uses an algorithm that computes the weak head normal form of a value. This algorithm is represented by a relation $u \Downarrow u'$ between values, which can be read as *u evaluates to* u'. This relation is inductively defined by:

– if $u \Downarrow (\lambda x\ M)\rho$, and $M(\rho, x = w) \Downarrow v$, then $u\ w \Downarrow v$,
– if $u \Downarrow u'$ and u' is a generic value or an application, then $u\ w \Downarrow u'\ w$,
– if $\mathsf{lookup}\ x\ \rho \Downarrow v$, then $x\rho \Downarrow v$,
– $\mathsf{Type}\ \rho \Downarrow \mathsf{Type}$,

– if $(M\rho)\ (N\rho) \Downarrow v$, then $(M\ N)\rho \Downarrow v$,
– $v \Downarrow v$ if v is of the form v_k or $M\rho$, where M is an abstraction or a product.

Notice that this relation is partial and deterministic. Furthermore,

Lemma 3 *If $u \Downarrow v$, then $u = v$.*

Corollary 1 *If $u \Downarrow v$, then $f\ u = f\ v \in \mathsf{D}$, for any model D, and any assignment $f \in \mathsf{G}{\rightarrow}\mathsf{D}$.*

2.3.2 Conversion

The conversion algorithm is represented by a relation $u_1 \sim u_2$. We define inductively that $u_1 \sim u_2$ holds if:

– $u_1 \Downarrow \mathsf{Type}$ and $u_2 \Downarrow \mathsf{Type}$, or
– $u_1 \Downarrow t_1\ w_1$, $u_2 \Downarrow t_2\ w_2$, and $t_1 \sim t_2$ and $w_1 \sim w_2$, or
– $u_1 \Downarrow v_{k_1}$, $u_2 \Downarrow v_{k_2}$ and $k_1 = k_2$, or
– $u_1 \Downarrow (\lambda x_1\ M_1)\rho_1$, $u_2 \Downarrow (\lambda x_2\ M_2)\rho_2$ and $M_1(\rho_1, x_1 = v_k) \sim M_2(\rho_2, x_2 = v_k)$ where v_k does not occur in ρ_1, ρ_2, or
– $u_2 \Downarrow ((x_1 : A_1)B_1)\rho_1$, $u_2 \Downarrow ((x_2 : A_2)B_2)\rho_2$ and we have both $A_1\rho_1 \sim A_2\rho_2$ and $B_1(\rho_1, x_1 = v_k) \sim B_2(\rho_2, x_2 = v_k)$ where v_k does not occur in ρ_1, ρ_2.

From Lemma 3, we get the semantical soundness of this algorithm.

Lemma 4 *If $u_1 \sim u_2$, then $u_1 = u_2$.*

Corollary 2 *If $u_1 \sim u_2$, then $f\ u_1 = f\ u_2$ for any model D and any assignment $f \in \mathsf{G}{\rightarrow}\mathsf{D}$.*

2.4 Type-Checking Algorithm

Ultimately, we want to check when an expression A is a correct type, and given such an expression A, when an expression M is of type A. As a technical intermediary notion, it is convenient to introduce a typing relation between expressions and values: M is of type u will mean that the value $M()$ is of type u. Recursively, we need to express when a given expression M, in a given pair of environments ρ and Γ, is of type u, where u is a value. Intuitively, ρ assigns values and Γ type values to the free variables of M. We write this relation $\rho;\Gamma \vdash M \Rightarrow v$. We need to define simultaneously the type inference relation $\rho;\Gamma \vdash M \mapsto v$, meaning that it is possible to infer the type value v for M in the environments ρ and Γ.

The type-checking algorithm $\rho;\Gamma \vdash M \Rightarrow v$ and the type inference algorithm

$\rho;\Gamma \vdash M \mapsto v$ are represented by two relations defined inductively and simultaneously.

- If $v \Downarrow ((y:A)B)\rho'$, and $\rho, x = v_k; \Gamma, x : A\rho' \vdash N \Rightarrow B(\rho', y = v_k)$ where v_k does not occur in ρ, Γ, ρ', then $\rho;\Gamma \vdash \lambda y\ N \Rightarrow v$,
- if $\rho;\Gamma \vdash A \Rightarrow \mathsf{Type}$, and $\rho, x = v_k; \Gamma, x : A\rho \vdash B \Rightarrow \mathsf{Type}$, where v_k does not occur in ρ, Γ, and $v \Downarrow \mathsf{Type}$, then $\rho;\Gamma \vdash (x:A)B \Rightarrow v$,
- if $\rho;\Gamma \vdash M \mapsto w$ and $w \sim v$, then $\rho;\Gamma \vdash M \Rightarrow v$,
- $\rho;\Gamma \vdash x \mapsto x\Gamma$, if x occurs in Γ,
- if $\rho;\Gamma \vdash M_1 \mapsto u_1$ and $u_1 \Downarrow ((x:A)B)\rho'$, and $\rho;\Gamma \vdash M_2 \Rightarrow A\rho'$, then $\rho;\Gamma \vdash M_1\ M_2 \mapsto B(\rho', x = M_2\rho)$,
- $\rho;\Gamma \vdash \mathsf{Type} \mapsto \mathsf{Type}$.

Proposition 2 *If* $\vdash A \Rightarrow \mathsf{Type}$ *and* $\vdash M \Rightarrow A()$, *then* $[\![A]\!] : \mathsf{Type}$ *and* $[\![M]\!] : [\![A]\!]$ *in* D.

Proof: We prove more generally, by simultaneous induction on the definition of the two relations of type-checking and type inference, that, for any assignment f such that $\mathsf{eval}\ x\ (f^*\ \rho)$ is of type $\mathsf{eval}\ x\ (f^*\ \Gamma)$ in D for all $x \in \mathsf{dom}(\Gamma)$, if $\rho;\Gamma \vdash M \Rightarrow u$ or $\rho;\Gamma \vdash M \mapsto u$, then $\mathsf{eval}\ M\ (f^*\ \rho)$ is of type $f\ u$ in D.

We illustrate only the abstraction case of the type-checking relation, more delicate than the other cases. We have to check that, for any suitable assignment f, the value $f\ ((\lambda x\ N)\rho)$ is of type $f\ v$, with $f\ v = \mathsf{eval}\ (y:A)B\ (f^*\ \rho')$. For this it is enough to check that $\mathsf{eval}\ N\ ((f^*\ \rho), x = d)$ is of type $\mathsf{eval}\ B\ ((f^*\ \rho'), y = d)$ for any $d \in \mathsf{D}$ which is of type $\mathsf{eval}\ A\ (f^*\ \rho')$.

Let g be the assignment that differs from f only on the variable v_k and such that $g\ v_k = d$. Because v_k does not occur in ρ, ρ', we have $\mathsf{eval}\ N\ ((f^*\ \rho), x = d) = \mathsf{eval}\ N\ (g^*\ (\rho, x = v_k))$ and $\mathsf{eval}\ B\ ((f^*\ \rho'), y = d) = \mathsf{eval}\ B\ (g^*\ (\rho', y = v_k))$, and $\mathsf{eval}\ A\ (f^*\rho') = \mathsf{eval}\ A\ (g^*\ \rho')$. Hence the result follows by induction hypothesis. $\square$

Notice that our algorithm accepts the following judgement

$$\vdash \lambda x \lambda x\ x : (x:A)(y:P\ x)P\ x,$$

which is not derivable in Martin-Löf's substitution calculus [16].

3 Extension to let expressions

This treatment extends directly to the addition of let expressions. We add expressions of the form $\mathsf{let}\ x = M : A\ \mathsf{in}\ N$. The meaning of this expression is reflected by the conversion rule

$$(\mathsf{let}\ x\ =\ M : A\ \mathsf{in}\ N)\rho = N(\rho, x = M\rho).$$

The typing rule is that $\rho; \Gamma \vdash \mathsf{let}\ x\ =\ M : A\ \mathsf{in}\ N \Rightarrow v$ iff $\rho; \Gamma \vdash A \Rightarrow \mathsf{Type}$ and $\rho; \Gamma \vdash M \Rightarrow A\rho$ and $\rho, x = M\rho; \Gamma, x : A\rho \vdash N \Rightarrow v$.

We get a language that is similar to de Bruijn's $\Lambda\Delta$ [5]. The problem of type-checking such let expressions is explained and motivated with a concrete example at the end of the survey article [4].

Related Works and Conclusion

We have presented a simple implementation and correctness proof of a type-checking algorithm for dependent types, while being explicit about the problem of α-conversion. This is made possible by the explicit introduction of closures.

Previous attempts of a complete description of a type-checking algorithm for dependent types can be found in [14,16,5]. In reference [1] there is presented a complete type-checking algorithm for second-order lambda-calculus, that contains most of the difficulty of type-checking dependent types. This algorithm has been used in the language Quest [6]. As can be seen by comparing this algorithm with the algorithm we present, our approach is more straightforward. A closer formalism is a predecessor of this work on explicit substitution, presented in [9], which introduces the idea of explicit closures.

We think that the same method can be used to simplify the presentation of the semantics of languages with a binding structure, and the meta-mathematical analysis of languages with dependent types [4].

Acknowledgement

This work owes much to several discussions with Dan Synek. Randy Pollack and Luca Cardelli made valuable comments on a preliminary version. Thanks to Bernhard Möller and to the referee for their comments concerning the presentation of this note.

References

[1] M.Abadi, L.Cardelli, P.L. Curien, J.J. Levy. Explicit Substitutions. Journal of Functional Programming 1 (4), p. 375-416, 1991.

[2] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1986.

[3] L. Augustsson. Haskell B. user's manual available over WWW from hhtp://www.cs.chalmers.se:80/pub/haskell/chalmers

[4] H. Barendregt. Lambda Calculi with Types. Handbook of Logic in Computer Science, Volume II, Edited by S. Abramski, D.M. Gabbai and T.S.E. Maibaum, Oxford University Press.

[5] N.G. de Bruijn. A plea for weaker frameworks. in Logical Framework, G. Huet and G. Plotkin eds, Cambridge University Press, p. 40 - 68, 1991.

[6] L. Cardelli. Typeful Programming. in: Formal Description of Programming Concepts, E.J. Neuhold,.Paul Eds., Springer-Verlag, 1991.

[7] Th. Coquand and G. Huet. The Calculus of Constructions. Information and Computation 76, p. 95 - 120, 1988.

[8] Th. Coquand and H. Herbelin. A-translation and looping combinators in pure type system. Journal of Functional Programming, 1994, vol 4, p. 77-88.

[9] P.L. Curien. An abstract framework for environment machines. Theoretical Computer Science 82, p. 389-402, 1991.

[10] M.J. Gordon, A.J. Milner and C.P. Wadsworth. Edinburgh LCF - a Mechanised Logic of Computation. Vol. 78, Lecture Notes in Computer Science, Springer-Verlag, 1979.

[11] K. Hanna, N. Daeche. Purely Functional Implementation of a Logic. Vol. 230, Lecture Notes in Computer Science, Springer-Verlag, p. 598-607, 1986.

[12] L. Henkin. Completeness in the theory of types. Journal of Symbolic Logic, 15, p. 81-91, 1950.

[13] J.R. Hindley and J. Seldin. *Introduction to Combinators and λ-calculus.* London Mathematical Society Student Texts 1, Cambridge University press, 1986.

[14] G. Huet. The Constructive Engine. In R. Narasimhan, editor, A Perspective in Theoretical Computer Science, World Scientific Publishing, 1989.

[15] L. Paulson. Isabelle: The Next 700 Theorem Provers. In: P. Odifreddi, ed., *Logic and Computer Science*, 1990, p. 361-386.

[16] R. Pollack. Closure under Alpha Conversion. In: Types for Proofs and Programs, H. Barendregt, T. Nipkow eds., Vol. 806, Lecture Notes in Computer Science, Springer-Verlag, p. 313 - 332, 1993.

[17] A. Stoughton. Substitution Revisited. Theoretical Computer Science 59 (1988), p. 317-325.

[18] J. Stoy. *Denotational Semantics.* MIT Press, Cambridge, 1977

[19] A. Tatsitro. Formulation of Martin-Löf's Theory of Types with Explicit Substitution. Licentiate Thesis, Chalmers University, 1993.

[20] G. Winskel. *The formal semantics of prorgamming languages, an introduction.* The MIT Press, 1993

Appendix: a Gofer/Haskell implementation

```
-- the main data types and general functions

type Id = String

data Exp =
 Var Id | App Exp Exp |  Abs Id Exp | Let Id Exp Exp Exp
 | Pi Id Exp Exp | Type

data Val = VGen Int | VApp Val Val | VType | VClos Env Exp

type Env = [(Id,Val)]

update :: Env -> Id -> Val -> Env
update env x u = (x,u):env

lookup :: Id -> Env -> Val
lookup x ((y,u):env) =
 if x == y then u
 else lookup x env
lookup x [] = error ("lookup " ++ x)

-- a short way of writing the whnf algorithm

app :: Val -> Val -> Val
eval :: Env -> Exp -> Val

app u v =
 case u of
   VClos env (Abs x e) -> eval (update env x v) e
   _ -> VApp u v

eval env e =
 case e of
   Var x -> lookup x env
   App e1 e2 -> app (eval env e1) (eval env e2)
   Let x e1 _ e3 -> eval (update env x (eval env e1)) e3
   Type -> VType
   _ -> VClos env e

whnf :: Val -> Val
whnf v =
 case v of
   VApp u w -> app (whnf u) (whnf v)
   VClos env e -> eval env e
   _ -> v

-- the conversion algorithm; the integer is
-- used to represent the introduction of a fresh variable

eqVal :: (Int,Val,Val) -> Bool
eqVal (k,u1,u2) =
 case (whnf u1,whnf u2) of
   (VType,VType) -> True
   (VApp t1 w1,VApp t2 w2) ->
      eqVal (k,t1,t2) && eqVal (k,w1,w2)
   (VGen k1,VGen k2) -> k1 == k2
   (VClos env1 (Abs x1 e1),VClos env2 (Abs x2 e2)) ->
      let v = VGen k
      in eqVal (k+1,
                VClos (update env1 x1 v) e1,
                VClos (update env2 x2 v) e2)
   (VClos env1 (Pi x1 a1 b1),VClos env2 (Pi x2 a2 b2)) ->
      let v = VGen k
```

```
      in eqVal (k,VClos env1 a1,VClos env2 a2) &&
         eqVal (k+1,
                VClos (update env1 x1 v) b1,
                VClos (update env2 x2 v) b2)
   _ -> False

-- type-checking and type inference

checkExp :: (Int,Env,Env) -> Exp -> Val -> Bool
inferExp :: (Int,Env,Env) -> Exp -> Val
checkType :: (Int,Env,Env) -> Exp -> Bool

checkType (k,rho,gamma) e = checkExp (k,rho,gamma) e VType

checkExp (k,rho,gamma) e v =
 case e of
   Abs x n ->
    case whnf v of
      VClos env (Pi y a b) ->
        let v = VGen k
        in checkExp (k+1,
                     update rho x v,
                     update gamma x (VClos env a))
                    n (VClos (update env y v) b)
      _ -> error"expected Pi"
   Pi x a b ->
     case whnf v of
       VType -> checkType (k,rho,gamma) a &&
               checkType (k+1,
                          update rho x (VGen k),
                          update gamma x (VClos rho a))
                         b
       _ -> error"expected Type"
   Let x e1 e2 e3 ->
     checkType (k,rho,gamma) e2 &&
     checkExp (k,
               update rho x (eval rho e1),
               update gamma x (eval rho e2))
              e3 v
   _ -> eqVal (k,inferExp (k,rho,gamma) e,v)

inferExp (k,rho,gamma) e =
 case e of
   Var id -> lookup id gamma
   App e1 e2 ->
      case whnf (inferExp (k,rho,gamma) e1) of
        VClos env (Pi x a b) ->
          if checkExp (k,rho,gamma) e2 (VClos env a)
               then VClos (update env x (VClos rho e2)) b
          else error"application error"
        _ -> error"application, expected Pi"
   Type -> VType
   _ -> error"cannot infer type"

typecheck :: Exp -> Exp -> Bool

typecheck m a =
 checkType (0,[],[]) a &&
 checkExp (0,[],[]) m (VClos [] a)

test :: Bool
test =
 typecheck (Abs "A" (Abs "x" (Var "x")))
           (Pi "A" Type (Pi "x" (Var "A") (Var "A")))
```