The Undecidability of Typability in the Lambda-Pi-Calculus

Gilles Dowek

The Undecidability of Typability in the Lambda-Pi-Calculus

Gilles Dowek

School of Computer Science, Carnegie Mellon University
Pittsburgh, PA 15213-3890, U.S.A.

Abstract. The set of pure terms which are typable in the $\lambda\Pi$-calculus in a given context is not recursive. So there is no general type inference algorithm for the programming language Elf and, in some cases, some type information has to be mentioned by the programmer.

Introduction

The programming language Elf [13] is an extension of λ-Prolog in which the clauses are expressed in a λ-calculus with dependent types ($\lambda\Pi$-calculus [8]). Since this calculus verifies the propositions-as-types principle, a proof of a proposition is merely a term of the calculus. Using this property of the $\lambda\Pi$-calculus, the programmer can either express a proposition and let the machine search for a proof of this proposition (as in usual logic programming) or express both a proposition and its proof and let the machine check that this proof is correct (as in proof-verification systems). Thus Elf can be used both to express logic programs and to reason about of their properties.

A *type inference algorithm* for a given language is an algorithm which assigns a type to each variable of a program. Thus, when such an algorithm exists, the types of the variables do not need to be mentioned by the programmer. As an example, a type inference algorithm for the language ML is given in [3].

We show here that the set of pure terms which are typable in the $\lambda\Pi$-calculus in a given context is not recursive. So there is no general type inference algorithm for the language Elf and, in some cases, some type information has to be mentioned by the programmer.

As already remarked in [3], typing a term requires the solution of a unification problem. Typing a term in the simply typed λ-calculus (and in ML) requires the solution of a first order unification problem and thus typability is decidable in these languages.

Typing a term in the $\lambda\Pi$-calculus requires the solution of a unification problem which is also formulated in the $\lambda\Pi$-calculus. Unification in the $\lambda\Pi$-calculus has been shown to be undecidable (third order unification in [9], then second order unification in [7] and third order pattern matching in [4]), i.e. there is no algorithm that decides if such a unification problem has a solution. But in order to prove the undecidability of typability in the $\lambda\Pi$-calculus we need to prove that there is no algorithm that decides if a unification problem *produced by a* *typing problem* has a solution. Unification problems produced by typing problems are very restricted and the undecidability proofs of unification have to be adapted to this class of problems. We show here that the proof of [9] can easily be adapted.

1 The Lambda-Pi-Calculus

We follow [1] for a presentation of the $\lambda\Pi$-calculus. The set of *terms* is inductively defined by

$$T \ ::= \ Type \mid Kind \mid x \mid (T\ T) \mid \lambda x : T.T \mid \Pi x : T.T$$

In this note, we ignore variable renaming problems. A rigorous presentation would use de Bruijn indices. The terms $Type$ and $Kind$ are called *sorts*, the terms x *variables*, the terms $(t\ t')$ *applications*, the terms $\lambda x : t.t'$ *abstractions* and the terms $\Pi x : t.t'$ *products.* The notation $t \to t'$ is used for $\Pi x : t.t'$ when x has no free occurrence in t'.

Let t and t' be terms and x a variable. We write $t[x \leftarrow t']$ for the term obtained by substituting t' for x in t. We write $t \cong t'$ when the terms t and t' are β-equivalent ($\beta\eta$-equivalence can also be considered without affecting the proof given here).

A *context* is a list of pairs $< x, T >$ (written $x : T$) where x is a variable and T a term.

We define inductively two judgements: Γ *is well-formed* and t *has type* T *in* Γ ($\Gamma \vdash t : T$) where Γ is a context and t and T are terms.

$$\frac{}{[\,]\ \text{well-formed}}$$

$$\frac{\Gamma \vdash T : s}{\Gamma[x : T]\ \text{well-formed}}\ s \in \{Type, Kind\}$$

$$\frac{\Gamma\ \text{well-formed}}{\Gamma \vdash Type : Kind}$$

$$\frac{\Gamma\ \text{well-formed} \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash T : Type \quad \Gamma[x : T] \vdash T' : s}{\Gamma \vdash \Pi x : T.T' : s}\ s \in \{Type, Kind\}$$

$$\frac{\Gamma \vdash \Pi x : T.T' : s \quad \Gamma[x : T] \vdash t : T'}{\Gamma \vdash \lambda x : T.t : \Pi x : T.T'}\ s \in \{Type, Kind\}$$

$$\frac{\Gamma \vdash t : \Pi x : T.T' \quad \Gamma \vdash t' : T}{\Gamma \vdash (t\ t') : T'[x \leftarrow t']}$$

$$\frac{\Gamma \vdash T : s \quad \Gamma \vdash T' : s \quad \Gamma \vdash t : T \quad T \cong T'}{\Gamma \vdash t : T'}\ s \in \{Type, Kind\}$$

A term t is said to be *well-typed* in a context Γ if there exists a term T such that $\Gamma \vdash t : T$.

The reduction relation on well-typed terms is strongly normalizable and confluent. Thus each well-typed term has a unique normal form and two terms are equivalent if they have the same normal form [8] ([6] [15] [2] if $\beta\eta$-equivalence is considered).

A term t well-typed in a context Γ has a unique type modulo equivalence.

A normal term t well-typed in a context Γ has either the form

$$t = \lambda x_1 : T_1....\lambda x_n : T_n.(x\ c_1\ ...\ c_n)$$

where x is a variable or a sort or

$$t = \lambda x_1 : T_1....\lambda x_n : T_n.\Pi x : P.Q$$

The *head symbol* of t is x is the first case and, by convention, the symbol Π in the second. The *top variables* of t are the variables $x_1, ..., x_n$.

2 Typability in the Lambda-Pi-Calculus

Definition 1. *A term t of type T in a context Γ is said to be an object in Γ if $\Gamma \vdash T : Type$.*

Proposition 1. *If a term t is an object in a context Γ then it is either a variable, an application or an abstraction. If it is an application $t = (u\ v)$ then both terms u and v are objects in Γ, if it is an abstraction $t = \lambda x : U.u$ then the term u is an object in the context $\Gamma[x : U]$.*

Definition 2. *The set of pure terms is inductively defined by*

$$T\ ::=\ x \mid (T\ T) \mid \lambda x.T$$

Definition 3. *Let t be an object in a context Γ, the content of t ($|t|$) is the pure term defined by induction over the structure of t by*
- $|x| = x$,
- $|(t\ t')| = (|t|\ |t'|)$,
- $|\lambda x : U.t| = \lambda x.|t|$.

A pure term t is said to be typable in a context Γ if there exists a term t' well-typed in an extension $\Gamma\Delta$ of Γ such that t' is an object in $\Gamma\Delta$ and $t = |t'|$.

Remark 1. Typing a pure term in a context Γ is assigning a type to bound variables and to some of the free variables, while the type of the other free variables is given in the context Γ. When the context Γ is empty, then typing a term in Γ is assigning a type to both bound and free variables.

Proposition 2. *Typability in the empty context is decidable in the $\lambda\Pi$-calculus.*

Proof. Pure terms typable in the empty context in the $\lambda\Pi$-calculus and in the simply typed λ-calculus are the same [8] and typability is decidable in simply typed λ-calculus [3].

3 Post Correspondence Problem

Definition 4. *Post Correspondence Problem*
A Post correspondence problem is a finite set of pairs of words over the two letters alphabet $\{A, B\}$ *:* $\{< \varphi_1, \psi_1 >, ..., < \varphi_n, \psi_n >\}$*. A solution to such a problem is a non empty sequence of integers* $i_1, ..., i_p$ *such that*

$$\varphi_{i_1} ... \varphi_{i_p} = \psi_{i_1} ... \psi_{i_p}$$

Theorem 1. *(Post [14]) It is undecidable whether or not a Post problem has a solution.*

4 Undecidability of Typability in the Lambda-Pi-Calculus

Let us consider the context
$\Gamma = [T : Type; a : T \to T; b : T \to T; c : T; d : T; P : T \to Type;$
$F : \Pi x : T.((P\ x) \to T)]$

Definition 5. *(Huet [9]) Let* φ *be a word in the two letters alphabet* $\{A, B\}$*, we define by induction on the length of* φ *the term* $\hat{\varphi}$ *well-typed in* Γ *and the pure term* $\tilde{\varphi}$ *as follows*

$$\hat{\varepsilon} = \lambda y : T.y$$

$$\hat{A\varphi} = \lambda y : T.(a\ (\hat{\varphi}\ y))$$

$$\hat{B\varphi} = \lambda y : T.(b\ (\hat{\varphi}\ y))$$

$$\tilde{\varphi} = |\hat{\varphi}|$$

Proposition 3. *Let* $\{< \varphi_1, \psi_1 >, ..., < \varphi_n, \psi_n >\}$ *be a Post problem, the non empty sequence* $i_1, ..., i_p$ *is a solution to this problem if and only if*

$$(\hat{\varphi_{i_1}}\ (...(\hat{\varphi_{i_p}}\ c)...)) \cong (\hat{\psi_{i_1}}\ (...(\hat{\psi_{i_p}}\ c)...))$$

Proposition 4. *If* g *is a term such that the term* $(g\ a\ ...\ a)$ *(*n *symbols* a*) is well-typed and is an object in an extension* $\Gamma\Delta$ *of* Γ *then the term* g *is well-typed in the context* $\Gamma\Delta$ *and its type is equivalent to the term*

$$\Pi x_1 : T \to T. ... \Pi x_n : T \to T.(\beta\ x_1\ ...\ x_n)$$

for some term β *of type* $(T \to T) \to ... \to (T \to T) \to Type$ *in the context* $\Gamma\Delta$*.*

Proof. By induction on n.

Proposition 5. *Let* $t, u_1, ..., u_n, v$ *be normal terms such that* $(t\ u_1\ ...\ u_n)$ *is a well-typed term and its normal form is* v*. The head symbol of the* t *is either the head symbol of* v *or a top variable of* t*.*

Proof. Let x be the head symbol of t. If x is not a top variable of t then the head symbol of the normal form of $(t\ u_1\ ...\ u_n)$ is also x, so x is the head symbol of v.

Proposition 6. *Let t be a normal term of type $(T \to T) \to ... \to (T \to T) \to T$ in the context Γ such that the normal form of $(t\ \lambda y : T.y\ ...\ \lambda y : T.y)$ is equal to c. Then the term t has the form*

$$t = \lambda x_1 : T \to T....\lambda x_n : T \to T.(x_{i_1}\ (...(x_{i_p}\ c)...))$$

for some sequence $i_1, ..., i_p$.

Proof. By induction on the number of variable occurrences in t.

Theorem 2. *It is undecidable whether or not a pure term is typable in a given context.*

Proof. Consider a Post problem $\{< \varphi_1, \psi_1 >, ..., < \varphi_n, \psi_n >\}$. We construct the pure term t such that t is typable in Γ if and only if the Post problem has a solution.

$$\begin{array}{ll} t = \lambda f.\lambda g.\lambda h.(f & (g\ a\ ...\ a) \\ & (h\ (g\ \tilde{\varphi_1}\ ...\ \tilde{\varphi_n})) \\ & (h\ (g\ \tilde{\psi_1}\ ...\ \tilde{\psi_n})) \\ & (F\ c\ (g\ \lambda y.y\ ...\ \lambda y.y)) \\ & (F\ d\ (g\ \lambda y.d\ ...\ \lambda y.d))) \end{array}$$

Assume this term is typable and call α the type of g. The term $(g\ a\ ...\ a)$ is well-typed and is an object in $\Gamma\Delta$ so

$$\alpha \cong \Pi x_1 : T \to T....\Pi x_n : T \to T.(\beta\ x_1\ ...\ x_n)$$

where β is a term of type $(T \to T) \to ... \to (T \to T) \to Type$ in $\Gamma\Delta$.

Then all the variables y bound in the terms $\tilde{\varphi}_i$, $\tilde{\psi}_i$, $\lambda y.y$ and $\lambda y.d$ have type T. The term $(g\ \hat{\varphi}_1\ ...\ \hat{\varphi}_n)$ has the type $(\beta\ \hat{\varphi}_1\ ...\ \hat{\varphi}_n)$, so from the well-typedness of the term $(h\ (g\ \hat{\varphi}_1\ ...\ \hat{\varphi}_n))$ we get that the type of the variable h has the form $\Pi x : \gamma.\gamma'$ and

$$\gamma \cong (\beta\ \hat{\varphi}_1\ ...\ \hat{\varphi}_n)$$

in the same way, from the well-typedness of the term $(h\ (g\ \hat{\psi}_1\ ...\ \hat{\psi}_n))$ we get

$$\gamma \cong (\beta\ \hat{\psi}_1\ ...\ \hat{\psi}_n)$$

so

$$(\beta\ \hat{\varphi}_1\ ...\ \hat{\varphi}_n) \cong (\beta\ \hat{\psi}_1\ ...\ \hat{\psi}_n)$$

From the well-typedness of the term $(F\ c\ (g\ \lambda y : T.y\ ...\ \lambda y : T.y))$ we get

$$(\beta\ \lambda y : T.y\ ...\ \lambda y : T.y) \cong (P\ c)$$

At last from the the well-typedness of the term $(F\ d\ (g\ \lambda y : T.d\ ...\ \lambda y : T.d))$ we get

$$(\beta\ \lambda y : T.d\ ...\ \lambda y : T.d) \cong (P\ d)$$

Since the term β has type $(T \to T) \to ... \to (T \to T) \to Type$, the head symbol of the normal form of the term β cannot be a top variable of β, so it is the variable P and we have

$$\beta \cong \lambda x_1 : T \to T....\lambda x_n : T \to T.(P\ (\delta\ x_1\ ...\ x_n))$$

For some term δ of type $(T \to T) \to ... \to (T \to T) \to T$. We get

$$(\delta\ \hat{\varphi}_1\ ...\ \hat{\varphi}_n) \cong (\delta\ \hat{\psi}_1\ ...\ \hat{\psi}_n)$$

$$(\delta\ \lambda y : T.y\ ...\ \lambda y : T.y) \cong c$$

$$(\delta\ \lambda y : T.d\ ...\ \lambda y : T.d) \cong d$$

The second equality shows that the normal form of the term δ has the form

$$\lambda x_1 : T \to T....\lambda x_n : T \to T.(x_{i_1}\ (...(x_{i_p}\ c)...))$$

for some sequence $i_1, ..., i_p$. The third equality shows that $p > 0$ and the first one that

$$(\hat{\varphi_{i_1}}\ (...(\hat{\varphi_{i_p}}\ c)...)) \cong (\hat{\psi_{i_1}}\ (...(\hat{\psi_{i_p}}\ c)...))$$

so the sequence $i_1, ..., i_p$ is a solution to the Post problem.

Conversely assume that the Post problem has a solution $i_1, ..., i_p$, then by giving the following types to the variables f, g and h

$$f : (P\ (a\ (...(a\ c)...))) \to T \to T \to T \to T \to T$$

$$g : \Pi x_1 : T \to T....\Pi x_n : T \to T.(P\ (x_{i_1}\ (...(x_{i_p}\ c)...)))$$

$$h : (P\ (\hat{\varphi_{i_1}}\ (...(\hat{\varphi_{i_p}}\ c)...))) \to T$$

and the type T to all the other variables of the term t, we get a term t' well-typed in Γ, which is an object and such that $t = |t'|$.

Remark 2. Along the way, we have proved that in the simply typed λ-calculus, the unification problems of the form

$$(f\ t_1\ ...\ t_n) = (f\ t'_1\ ...\ t'_n)$$

$$(f\ u_1\ ...\ u_n) = u'$$

$$(f\ v_1\ ...\ v_n) = v'$$

where $t_i, t'_i, u_i, u', v_i, v'$ are closed terms and f a third order variable are undecidable.

It is decidable if each of these equations has a solution or not (since the first one is flexible-flexible [10] [11] and the others third order matching problems [5]), but it is undecidable whether or not they have a solution *in common.* If the variable f is second order the problems of this form are decidable since the second order matching algorithm [11] [12] produces a finite complete set of closed solutions.

Acknowledgements

The author thanks Frank Pfenning for many stimulating and helpful discussions on this problem and Pawel Urzyczyn for his careful reading of a previous draft of this paper.

References

1. H. Barendregt, Introduction to Generalized Type Systems, *Journal of Functional Programming* **1, 2** (1991) 125–154.
2. Th. Coquand, An Algorithm for Testing Conversion in Type Theory, *Logical Frameworks*, G. Huet and G. Plotkin (Eds.), Cambridge University Press (1991).
3. L. Damas, R. Milner, Principal Type-Scheme for Functional Programs, *Proceedings of Principles of Programming Languages* (1982).
4. G. Dowek, L'Indécidabilité du Filtrage du Troisième Ordre dans les Calculs avec Types Dépendants ou Constructeurs de Types (The Undecidability of Third Order Pattern Matching in Calculi with Dependent Types or Type Constructors), *Comptes Rendus à l'Académie des Sciences* **I, 312, 12** (1991) 951–956.
5. G. Dowek, Third Order Matching is Decidable, *Proceedings of Logic in Computer Science* (1992) 2–10.
6. H. Geuvers, The Church-Rosser Property for $\beta\eta$-reduction in Typed Lambda Calculi, *Proceedings of Logic in Computer Science* (1992) 453–460.
7. W.D. Goldfarb, The Undecidability of the Second-Order Unification Problem, *Theoretical Computer Science* **13** (1981) 225–230.
8. R. Harper, F. Honsell, G. Plotkin, A Framework for Defining Logics, *Proceedings of Logic in Computer Science* (1987) 194–204.
9. G. Huet, The Undecidability of Unification in Third Order Logic, *Information and Control* **22** (1973) 257–267.
10. G. Huet, A Unification Algorithm for Typed λ-calculus, *Theoretical Computer Science* **1** (1975) 27–57.
11. G. Huet, Résolution d'Équations dans les Langages d'Ordre 1, 2, ..., ω, *Thèse de Doctorat d'État*, Université de Paris VII (1976).
12. G. Huet, B. Lang, Proving and Applying Program Transformations Expressed with Second Order Patterns, *Acta Informatica* **11** (1978) 31–55.
13. F. Pfenning, Logic Programming in the LF Logical Framework, *Logical Frameworks*, G. Huet and G. Plotkin (Eds.), Cambridge University Press (1991).
14. E. L. Post, A Variant of a Recursively Unsolvable Problem, *Bulletin of American Mathematical Society* **52** (1946) 264–268.
15. A. Salvesen, The Church-Rosser Theorem for Pure Type Systems with $\beta\eta$-reduction, Manuscript, University of Edinburgh (1991).