# Type Inference for First-Order Logic

Aleksy Schubert*

Institute of Informatics
Warsaw University
ul. Banacha 2
02–097 Warsaw
Poland
alx@mimuw.edu.pl

**Abstract.** Although type inference for dependent types is in general undecidable, people experience that the algorithms for type inference in Elf programming language stop in common cases. The present paper is a partial explanation of this behaviour. It shows that for a wide range of terms — terms that correspond to first-order logic proofs — the formalism of dependent types gives decidable type inference. We remark also that imposing that the context and the type of a judgement are first-order is not sufficient for obtaining decidability.

## 1 Introduction

Lambda calculus with dependent types is a formalism defined in [HHP87] in order to provide a means for defining logics. For example, one can define first-order logic within the formalism. This definition leads to a restriction on dependent types which constitutes by itself an interesting type system for $\lambda$-terms.

Dependent types formalism has also been used as a base for the programming language Elf [Pfe91]. The clauses of Elf are expressions of dependent types. This allows to reason about properties of programs inside the language. Although the problem of inferring types in the language is undecidable, as shown in [Dow93], it comes out that for many practical programs the algorithm used in the framework halts. This paper is a partial explanation for the phenomenon. The type inference for a wide range of terms: terms that correspond to proofs in first-order logic, is decidable.

Interestingly enough, the border-line between decidability and undecidability is very slight here. The problem of type inference for the first-order logic inside dependent types is defined as follows: given a first-order context $\Gamma$ and a Curry-term $M$, check if there exists a first-order type such that $\Gamma \vdash M : \tau$ is the end of some first-order derivation (i.e. a derivation that can be translated into first-order logic). If the condition that $\Gamma \vdash M : \tau$ is the end of some first-order derivation is relaxed so that $\Gamma \vdash M : \tau$ may be the end of any derivation in

* This work was supported by Polish national research funding agency KBN grant no. 8 T11C 035 14.

J. Tiuryn (Ed.): FOSSACS 2000, LNCS 1784, pp. 297–313, 2000.

dependent types, then the problem becomes undecidable. This holds even for a class of terms $M$ that fall within an extension of the first-order logic where quantification over first-order function symbols is allowed.

The techniques used in this paper are based on the old idea that typing problems correspond to problems of solving appropriate equations. We reduce type inference for first-order logic to special kind of equations with explicit substitutions. These equations are subject to a further reduction that is very similar to usual Robinson's unification. So obtained equations are translated in turn to second-order unification equations. As it is proved in [Gol81], second-order unification is undecidable, so in general it cannot serve as a method for providing decidability. In the present study, we get a particular form of equations which allows us to design a procedure to solve them. We deal only with equations of the three forms

$$F_1(t_1,\dots,t_n) = F_2(s_1,\dots,s_m); \qquad F_1(t_1,\dots,t_n) = f(s_1,\dots,s_m);$$
$$f_1(t_1,\dots,t_n) = f_2(s_1,\dots,s_m)$$

where none of second-order variables may occur in terms $t_1,\dots,t_n,s_1,\dots s_n$. The latter condition is very important here as when we drop it the problem becomes undecidable. In [Sch98], it is shown that already solving equations f the form $F_1(t_1,\dots,t_n) = f(s_1,\dots,s_m)$, where second-order variables may occur in $s_1,\dots,s_n$, is undecidable.

The paper is organised as follows: Section 2 contains basic definitions and formulation of problems we deal with; Section 3 presents a sketch of the undecidability result for type inference with relaxed first-order constraints, and Section 4 contains a sketch of the decidability result for the first-order type inference.

## 2 Basic Definitions

We introduce the definition of the system $\lambda P$. The basic insight behind this system is that types of terms may depend on terms. ¿From the perspective of programming language this corresponds to providing devices for defining types such as `list(n)` that represent lists of length $n$. From the point of view off logic, this allows to implement rules of substitution. We follow hereafter the presentation in [SU98].

### 2.1 Language of $\lambda P$

The set of *pure* $\lambda$*-terms* is defined according to the following grammar:

$$M ::= x \mid (\lambda x.M) \mid (MM)$$

*Contexts* are used in the typing system as well. These are sequences of pairs: $(\alpha : \boldsymbol{\kappa})$ or $(x : \tau)$, where $\alpha$ is a kind variable, $\boldsymbol{\kappa}$ a kind, $x$ is an object variable and $\tau$ a type.

Pure $\lambda$-terms and contexts form a base for Curry style $\lambda$-calculus, $\lambda P$, the expressions of which are inferred according to the following rules:

### 2.2 Rules of $\lambda P$

*Kind formation rules:*

$$(\textit{type})\ \Gamma \vdash * : \Box \qquad (\text{kind-abs})\frac{\Gamma, x : \tau \vdash \boldsymbol{\kappa} : \Box}{\Gamma \vdash (\Pi x : \tau)\boldsymbol{\kappa} : \Box}$$

*Kinding rules:*

$$(\text{kind-var})\frac{\Gamma \vdash \boldsymbol{\kappa} : \Box}{\Gamma, \alpha : \boldsymbol{\kappa} \vdash \alpha : \boldsymbol{\kappa}}(\alpha \notin \mathrm{Dom}(\Gamma))$$

$$(\text{kind-app})\frac{\Gamma \vdash \phi : (\Pi x : \tau)\boldsymbol{\kappa} \quad \Gamma \vdash M : \tau}{\Gamma \vdash \phi M : \boldsymbol{\kappa}[x := M]}\ (\text{kind-abs})\frac{\Gamma, x : \tau \vdash \sigma : *}{\Gamma \vdash (\forall x : \tau)\sigma : *}$$

*Typing rules:*

$$(\text{var})\frac{\Gamma \vdash \tau : *}{\Gamma, x : \tau \vdash x : \tau}(\alpha \notin \mathrm{Dom}(\Gamma))$$

$$(\text{app})\frac{\Gamma \vdash N : (\forall x : \tau)\sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash NM : \sigma[x := M]}\ (\text{abs})\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x.M : (\forall x : \tau)\sigma}$$

*Weakening rules:*

$$(\text{trm} - \text{kd})\frac{\Gamma \vdash \tau : * \quad \Gamma \vdash \boldsymbol{\kappa} : \Box}{\Gamma, x : \tau \vdash \boldsymbol{\kappa} : \Box}(x \notin \mathrm{Dom}(\Gamma))$$

$$(\text{typ} - \text{kd})\frac{\Gamma \vdash \boldsymbol{\kappa} : \Box \quad \Gamma \vdash \boldsymbol{\kappa}' : \Box}{\Gamma, \alpha : \boldsymbol{\kappa} \vdash \boldsymbol{\kappa}' : \Box}(\alpha \notin \mathrm{Dom}(\Gamma))$$

$$(\text{trm} - \text{typ})\frac{\Gamma \vdash \tau : * \quad \Gamma \vdash \phi : \boldsymbol{\kappa}}{\Gamma, x : \tau \vdash \phi : \boldsymbol{\kappa}}(x \notin \mathrm{Dom}(\Gamma))$$

$$(\text{typ} - \text{typ})\frac{\Gamma \vdash \boldsymbol{\kappa} : \Box \quad \Gamma \vdash \phi : \boldsymbol{\kappa}'}{\Gamma, \alpha : \boldsymbol{\kappa} \vdash \phi : \boldsymbol{\kappa}'}(\alpha \notin \mathrm{Dom}(\Gamma))$$

$$(\text{trm} - \text{trm})\frac{\Gamma \vdash \tau : * \quad \Gamma \vdash M : \sigma}{\Gamma, x : \tau \vdash M : \sigma}(x \notin \mathrm{Dom}(\Gamma))$$

$$(\text{typ} - \text{trm})\frac{\Gamma \vdash \boldsymbol{\kappa} : \Box \quad \Gamma \vdash M : \sigma}{\Gamma, \alpha : \boldsymbol{\kappa} \vdash M : \sigma}(\alpha \notin \mathrm{Dom}(\Gamma))$$

*Conversion rules:*

$$(\text{kd} - \text{conv})\frac{\Gamma \vdash \phi : \boldsymbol{\kappa} \quad \boldsymbol{\kappa} =_\beta \boldsymbol{\kappa}'}{\Gamma \vdash \phi : \boldsymbol{\kappa}'}\ (\text{typ} - \text{conv})\frac{\Gamma \vdash M : \sigma \quad \sigma =_\beta \sigma'}{\Gamma \vdash M : \sigma'}$$

The definitions in Subsection 2.2 and 2.1 allow to infer types for Curry-style terms. The system $\lambda P$ is usually defined in the Church version as follows. First, raw expressions are described according to the following grammar:

$$\begin{array}{l}\Gamma ::= \{\} \mid \Gamma, (x : \phi) \mid \Gamma, (\alpha : \kappa);\\ \kappa ::= * \mid (\Pi x : \phi)\kappa;\\ \phi ::= \alpha \mid (\forall x : \phi)\phi \mid (\phi(M);\\ M ::= x \mid (MM) \mid (\lambda x : \phi.M)\end{array}$$

Some of these expressions, designated by inference rules, are called Church terms. The inference rules have exactly the same form in most cases as the rules for inferring Curry types. The exception is the rule (abs) which looks as follows

$$(\text{abs})\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau.M : (\forall x : \tau)\sigma}$$

Note that the definition of types for the Church version also changes — the definition depends on the definition of terms which is different in Curry and Church styles. These versions are essentially equivalent as shown in [vBLRU97].

In this document, we use the word 'subtype' to mean subexpression. We do not employ any other kind of 'subtyping' relation here.

**Definition 1** (type assignment)
A *derivation* is a tree labelled with rules of $\lambda P$ so that for each node its label premises are in bijection with conclusions in labels of its sons. Derivations are usually denoted by letters like $\mathcal{P}, \mathcal{Q} \ldots$

We say that *a derivation $\mathcal{P}$ assigns a type $\tau$ to a term $M$* iff the derivation ends with the assertion $\Gamma \vdash M : \tau$ for some $\Gamma$.

Except where stated explicitly otherwise, we write $(\forall x : \sigma)\tau$ as $\sigma \to \tau$ provided that $x$ does not occur free in $\tau$.

We shall be using extensively the notation $l((\forall x : \sigma_1)\sigma_2)$ and $l(\sigma_1 \to \sigma_2)$ to denote $\sigma_1$ together with $r((\forall x : \sigma_1)\sigma_2)$ and $r(\sigma_1 \to \sigma_2)$ to denote $\sigma_2$.

We denote the $\alpha$-conversion relation, applied for both types or $\lambda$-terms by $\equiv_\alpha$.

We have to formulate the exact problem we should solve. The notion of signature is central in the syntactic part in any presentation of the first-order logic. Thus, we have to determine what part of $\lambda P$ syntax corresponds to the notion.

**Definition 2** (signature first-order context)
The *signature first-order context* is a $\lambda$P context such that:

1. There is only one type variable 0 (which should be regarded as a type constant), representing the type individuals;
2. All kinds are of the form $0 \Rightarrow \cdots \Rightarrow 0 \Rightarrow *$;
3. There is a finite number of distinguished constructor variables, representing relation symbols in the signature (they must be of appropriate kinds, depending on arity);

4. Function symbols in the signature are represented by distinguished object variables of types $0 \to \cdots \to 0 \to 0$, depending on arity;
5. Constant symbols are represented by distinguished object variables of type 0.

A $\lambda P$ context obtained from a signature $\Sigma$ is denoted by $\Gamma_\Sigma$.

The proof theory for first-order logic introduces a notion of a context (environment) in which a formula is interpreted. This context is reflected by the following notion (this notion is in the spirit of [SU98]):

**Definition 3** (first-order context)
A *first-order context* over a signature context $\Gamma_\Sigma$ is a context in dependent types of the form $\Gamma_\Sigma \cup \{x_1 : \phi_1, \ldots, x_n : \phi_n\}$ where each $\phi_i$ is either first-order type or 0.

The notion of an algebraic term is crucial in the presentation of the first-order logic. These terms have their counterparts in $\lambda P$. The most straightforward definition of such terms looks as follows:

**Definition 4** (homogeneous first-order term)
We say that $t$ is a *homogeneous first-order term* in a first-order context $\Gamma$ iff

- $t = x$ where $\Gamma(x) = 0$ (i.e. $x$ is a constant symbol or a first-order variable),
- $t = ft_1 \ldots t_n$ where $\Gamma(f) = \underbrace{0 \to \cdots \to 0}_{n-times} \to 0$ and each $t_i$ is a homogeneous first-order term in $\Gamma$.

The next step in our presentation is to define what is the equivalent of the first-order formula.

**Definition 5** (first-order type)
We say that a type $\phi$ is a *first-order type* in the context $\Gamma$ iff it is of the form

- $P(t_1, \ldots, t_n)$ where $\Gamma(P) = \underbrace{0 \Rightarrow \cdots \Rightarrow 0}_{n-times} \Rightarrow *$, $P \neq 0$ and each $t_i$ is homogeneous first-order term in $\Gamma$, or
- $(\forall x_1 : 0) \cdots (\forall x_n : 0).\phi_1 \to \cdots \to \phi_m$ where each $\phi_i$ is a first-order type in $\Gamma \cup \{x_1 : 0, \ldots, x_n : 0\}$.

At last, we have to define which derivations of $\lambda P$ may be regarded as first-order derivations.

**Definition 6** (first-order derivations)
We say that a derivation $\mathcal{P}$ in dependent types is a *first-order derivation* iff each judgement $\Gamma \vdash M : \tau$ in the derivation is such that $\Gamma$ is a first-order context over a fixed signature first-order context $\Gamma_\Sigma$ and $\tau$ is either a first-order type or a type of the form $\underbrace{0 \to \cdots \to 0}_{n-times} \to 0$ where $n \geq 0$.

This definition of derivation allows to introduce one-to-one correspondence between derivations in $\lambda P$ and some first-order logic proofs. We do not present details due to limited space.

We can now describe precisely the set of problems we deal with.

*Problem 1.* The problem of *type inference with a first-order context* is defined as follows:
Given: A Curry-style term $M$ and a first-order context $\Gamma$.
Question: Does there exist a derivation in $\lambda P$ that ends with $\Gamma \vdash M : \tau$ for a first-order type $\tau$?

*Problem 2.* The problem of *first-order type inference* is defined as follows:
Given: A Curry-style term $M$ and a first-order context $\Gamma$.
Question: Does there exist a first-order derivation that ends with $\Gamma \vdash M : \tau$ for a first-order type $\tau$?

Note that in all the above-mentioned questions we assume that a context $\Gamma$ is a first-order context. This is not standard for type inference problems as usually these are formulated with arbitrary contexts. We may assume that contexts from the *Given* parts in definitions of problems are first-order contexts as procedure checking if the context has the property is easy.

## 3 Undecidability of Type Inference with a First-Order Context

Our undecidability proof is almost identical to the proof presented in [Dow93].

**Theorem 1.** *undecidability of Problem 1 The problem of type inference with a first-order context is undecidable.*

*Proof.* We present a description of changes that should be made in Dowek's proof in order to get our claim. The context $\Gamma$

$$[0 : \mathrm{Type}; a : 0 \to 0; b : 0 \to 0; c : 0; d : 0; P : 0 \to \mathrm{Type}; F : (\forall x : 0)(Px) \to 0]$$

used in [Dow93] should be replaced by

$$[0 : \mathrm{Type}; a : 0 \to 0; b : 0 \to 0; c : 0; d : 0; P : 0 \to \mathrm{Type}; F : (\forall x : 0)(Px) \to (Pc)].$$

The replacement enforces only a little change in types used in the proof (some occurrences should be replaced by $P(c)$), but this does not harm reasonings in Dowek's proof.

The undecidability is essentially obtained because we can quantify over first-order function symbols here. The type $\forall x_1 : 0 \to 0 \cdots \forall x_n : 0 \to 0.(\beta x_1 \cdots x_n)$ used in the proof is the only element that goes beyond first-order logic.

## 4 First-Order Type Inference

This material presents a proof for decidability of first-order type inference where signatures have at least one constant symbol. Such signatures give rise to a very wide class of instances and the restriction does not seem to be significant. In fact, the construction mentioned here requires only some minor modifications in order to provide a solution for the full problem. We lay aside the most general presentation for the sake of simplicity.

### 4.1 Generation of Equations

In our algorithm, we use some equations. Thus, we have to define the entities to be equated.

**Definition 7** (e-terms)
The set of *e-terms over the signature $\Sigma$ and variables $X$*, denoted by $T^e_\Sigma(X)$, is defined as follows

- $x \in T^e_\Sigma(X)$ if $x \in X$;
- $f(t_1, \ldots, t_n) \in T^e_\Sigma(X)$ if $f \in \Sigma$, has arity $n$ ($n \geq 0$) and $t_i \in T^e_\Sigma(X)$ for $i = 1, \ldots, n$;
- $t\langle x := s\rangle \in T^e_\Sigma(X)$ where $t \in T^e_\Sigma(X)$, $x \in X$, and $s$ is an e-term over $\Sigma$ with variables from $X$.

We extend this notion to types using a set $\mathcal{X}$ of type variables.

**Definition 8** (e-types)
The set of *e-types over the signature $\Sigma$, variables $X$ and type variables $\mathcal{X}$*, denoted by $\mathcal{T}^e_\Sigma(X, \mathcal{X})$, is defined as follows

- $P(t_1, \ldots, t_n) \in \mathcal{T}^e_\Sigma(X, \mathcal{X})$, if $t_i \in T^e_\Sigma(X)$ for $i = 1, \ldots, n$;
- $\alpha \in \mathcal{T}^e_\Sigma(X, \mathcal{X})$, if $\alpha \in \mathcal{X}$;
- $\tau_1 \rightarrow \tau_2 \in \mathcal{T}^e_\Sigma(X, \mathcal{X})$, if $\tau_1, \tau_2 \in \mathcal{T}^e_\Sigma(X, \mathcal{X})$ and $x \in X$;
- $(\forall x : 0)\tau \in \mathcal{T}^e_\Sigma(X, \mathcal{X})$, if $\tau \in \mathcal{T}^e_\Sigma(X, \mathcal{X})$ and $x \in X$;
- $\tau\langle x := s\rangle \in \mathcal{T}^e_\Sigma(X, \mathcal{X})$, if $\tau \in \mathcal{T}^e_\Sigma(X, \mathcal{X})$ and $s$ is a homogeneous first-order term over $\Sigma$ with variables from $X$.

We use the notation $\tau\langle \boldsymbol{x} := \boldsymbol{t}\rangle$ in order to shorten $\tau\langle x_1 := t_1\rangle \cdots \langle x_n := t_n\rangle$. The set of *free first-order variables* in an e-type (e-term) $\tau$, denoted by $\mathrm{Vars}(\tau)$, is defined so that $x$ is bounded in $(\forall x)\tau'$ and $\tau\langle x := t\rangle$.

We introduce the notation $\mathrm{TV}(\tau)$ to denote the set of all type variables in $\tau$. Notations $\mathrm{Vars}(\cdots)$ and $\mathrm{TV}(\cdots)$ are extended so that they can be applied to sets of e-types.

The notion of (semantical) substitution is not straightforward here so we present its definition. This notion describes a different operation than the one defined later in Definition 13.

**Definition 9** (first-order substitution)
A *first-order* substitution is a partial function from first-order variables to e-terms with finite domain. We usually denote such a substitution by $[x_1 := t_1, \ldots, x_n := t_n]$. This function acts on e-terms so that no free variable gets bounded, which may be expressed as follows:

- $x_i[x_1 := t_1, \ldots, x_i := t_i, \ldots, x_n := t_n] = t_i$;
- $y[x_1 := t_1, \ldots, x_n := t_n] = y$ where for each $i = 1, \ldots, n$ we have $x_i \neq y$;
- $f(s_1, \ldots, s_n)[x_1 := t_1, \ldots, x_n := t_n] = f(s'_1, \ldots, s'_n)$ where $s'_i = s_i[x_1 := t_1, \ldots, x_n := t_n]$;
- $t\langle x := u\rangle[x_1 := t_1, \ldots, x_n := t_n] = t'\langle x' := u'\rangle$ where $u' = u[x_1 := t_1, \ldots, x_n := t_n]$, the variable $x'$ does not occur in any of terms $x_1, \ldots, x_n$, $t_1, \ldots, t_n$ and $t' = t[x := x'][x_1 := t_1, \ldots, x_n := t_n]$;

- $\alpha[x_1 := t_1, \ldots, x_n := t_n] = \alpha$;
- $P(s_1, \ldots, s_n)[x_1 := t_1, \ldots, x_n := t_n] = P(s'_1, \ldots, s'_n)$ where $s'_i = s_i[x_1 := t_1, \ldots, x_n := t_n]$;
- $\tau_1 \to \tau_2[x_1 := t_1, \ldots, x_n := t_n] = \tau'_1 \to \tau'_2$ where $\tau'_i = \tau_i[x_1 := t_1, \ldots, x_n := t_n]$;
- $((\forall x : 0)\tau)[x_1 := t_1, \ldots, x_n := t_n] = ((\forall x' : 0)\tau')$ where $x'$ does not occur in $x_1, \ldots, x_n, t_1, \ldots, t_n$ and $\tau' = \tau[x := x'][x_1 := t_1, \ldots, x_n := t_n]$;
- $\tau\langle x := u\rangle[x_1 := t_1, \ldots, x_n := t_n] = \tau'\langle x' := u'\rangle$ where $u' = u[x_1 := t_1, \ldots, x_n := t_n]$, the variable $x'$ does not occur in any of terms $x_1, \ldots, x_n$, $t_1, \ldots, t_n$ and $\tau' = \tau[x := x'][x_1 := t_1, \ldots, x_n := t_n]$.

The set Paths($\tau$) of *paths* in an e-type $\tau$ (an e-term) is a set of sequences of natural numbers defined so that subsequent numbers represent which part of an e-type or an e-term is taken. For instance, the path 12 points to $\tau_2$ in $(\forall x)(\tau_1 \to \tau_2)$, and the path 3 points to $t$ in $\tau'\langle x := t\rangle$.

We have already introduced types with explicit substitutions (e-types). These substitutions allow to delay some substitution until a type variable is substituted, but then substitutions must be applied. The whole just described work is done by $\rightsquigarrow$-reduction.

**Definition 10** (reduction for e-terms and e-types)
The *reduction for e-terms* is defined as:

1. $t_1 = f(s_1, \ldots, s_n) \rightsquigarrow f(s'_1, \ldots, s'_n)$ where for some $i \in \{1, \ldots n\}$ we have $s_i \rightsquigarrow s'_i$ and for $j \neq i$ we have $s_j = s'_j$;
2. $t\langle x := s\rangle \rightsquigarrow t'\langle x := s\rangle$ when $t \rightsquigarrow t'$;
3. $t_1 = t\langle x := s\rangle \rightsquigarrow t[x := s]$ when $t$ is irreducible ($[x := s]$ is the usual substitution);
4. $\tau_1 = P(s_1, \ldots, s_n)$ and $\tau_2 = P(s'_1, \ldots, s'_n)$ for some predicate $P \in \Sigma$ and for some $i \in \{1, \ldots n\}$ we have $s_i \rightsquigarrow s'_i$ and for $j \neq i$ we have $s_j = s'_j$;
5. $\tau_1 = \sigma_1 \to \sigma_2$ and $\tau_2 = \sigma'_1 \to \sigma_2$ where $\sigma_1 \rightsquigarrow \sigma'_1$;
6. $\tau_1 = \sigma_1 \to \sigma_2$ and $\tau_2 = \sigma_1 \to \sigma'_2$ where $\sigma_2 \rightsquigarrow \sigma'_2$;
7. $\tau_1 = (\forall y : 0)\sigma_1$ and $\tau_2 = (\forall y : 0)\sigma'_1$ where $\sigma_1 \rightsquigarrow \sigma'_1$;
8. $\tau_1 = \sigma\langle x := s\rangle$ and $\tau_2 = \sigma'\langle x := s\rangle$, where $s \in T^e_\Sigma(X)$, $x \in X$, and $\sigma \rightsquigarrow \sigma'$;
9. $\tau_1 = (\sigma_1 \to \sigma_2)\langle x := s\rangle$ and $\tau_2 = \sigma_1\langle x := s\rangle \to \sigma_2\langle x := s\rangle$, where $s \in T^e_\Sigma(X)$, $x \in X$;
10. $\tau_1 = ((\forall y : 0)\sigma)\langle x := s\rangle$ and $\tau_2 = ((\forall y : 0)\sigma\langle x := s\rangle)$, where $s \in T^e_\Sigma(X)$, $x \neq y$ (if $x = y$ perform $\alpha$-conversion first and then reduce according to the present rule).
11. $\tau_1 = P(t_1, \ldots, t_m)\langle x := s\rangle$ and $\tau_2 = P(t'_1, \ldots, t'_m)$, where $s \in T^e_\Sigma(X)$, $x \in X$, $P \in \Sigma$ and has the arity $m$, and $t'_i = t_i\langle x := s\rangle$ for $i = 1, \ldots, m$.

As usual, we extend $\rightsquigarrow$ to its reflexive-transitive closure $\rightsquigarrow^*$.

We point out that according to the definition above an e-type of the form $\alpha\langle x := t\rangle$, where $\alpha$ is a variable, is irreducible.

The reduction $\rightsquigarrow^*$ has several good properties the proofs of which are omitted here: it has Church-Rosser property, it is strongly normalising, and decidable.

Thus, we can define that $NF^{\rightsquigarrow}(\tau)$ and $NF^{\rightsquigarrow}(t)$ which are normal forms of respectively the e-type $\tau$ and the e-term $t$.

The following interesting fact gives a nice insight about what is going on in e-types.

*Property 1.* If $\tau$ is an e-type in the normal form such that $\mathrm{TV}(\tau) = \emptyset$ then it has no subtype (subterm) of the form $\sigma\langle x := t\rangle$ ($s\langle x := t\rangle$).

**Definition 11** (equality for e-terms and e-types)
The *equality for e-terms and e-types* is defined as the least congruence containing $\rightsquigarrow^* \cup \equiv_\alpha$ and is denoted by $\simeq$.

**Definition 12** (e-equation)
We write $\tau_1 \doteq \tau_2$ to denote that the pair of e-types $\tau_1, \tau_2$ is an *e-equation*. Sets of e-equations are denoted by $\mathcal{E}, \mathcal{F}, \ldots$. The set $\mathcal{E}^e_\Sigma(X, \mathcal{X})$ is the set of e-equations among e-types from $\mathcal{T}^e_\Sigma(X, \mathcal{X})$.

Now, we define a notion of substitution we deal with.

**Definition 13** (substitutions)
Each partial function from type variables to some $\mathcal{T}^e_\Sigma(X, \mathcal{Y})$ is called a *substitution*.

We extend a substitution $S : \mathcal{X} \rightharpoonup \mathcal{T}^e_\Sigma(X, \mathcal{Y})$ to e-types inductively as follows:

- $S(0) = 0$;
- $S(P(t_1, \ldots, t_m)) = P(t_1, \ldots, t_m)$;
- $S(\alpha) = \alpha$ if $\alpha \notin \mathrm{Dom}(S)$;
- $S(\alpha) = S(\alpha)$ if $\alpha \in \mathrm{Dom}(S)$;
- $S((\forall x : 0)\sigma_2) = (\forall x : 0)S(\sigma_2)$;
- $S(\sigma_1 \to \sigma_2) = S(\sigma_1) \to S(\sigma_2)$;
- $S(\sigma\langle x := s\rangle) = S(\sigma)\langle x := s\rangle$.

Note that in the definition above we do not have any kind of renaming of individual variables while substituting under quantifier. This approach is intentional here. We agree with the fact that some symbols may get bounded during such a substitution.

**Definition 14** (solution of a set of equations)
We say that a substitution $S : \mathcal{X} \rightharpoonup \mathcal{T}^e_\Sigma(X, \emptyset)$ is a *solution of a set of e-equations* $\mathcal{E}$ iff for each e-equation $[\tau_1 \doteq \tau_2] \in \mathcal{E}$ we have $S(\tau_1) \simeq S(\tau_2)$.

We define $S(\Gamma)^{\rightsquigarrow}$ for a context $\Gamma$ as the sequence $\Gamma$ with each $x : \tau$ replaced by $x : NF^{\rightsquigarrow}(S(\tau))$.

We cannot hope for a most-general solution property here.

*Example 1.* Consider a signature context $\Gamma_\Sigma = \{P : 0 \Rightarrow *, c : 0\}$. The set of equations $\mathcal{E} = \{\alpha\langle x := c\rangle\langle y := c\rangle \doteq P(c)\}$ has two solutions $S_1(\alpha) = P(x)$ and $S_2(\alpha) = P(y)$, but there is no $S_3$ such that $S_3 \circ S_2 = S_1$ or $S_3 \circ S_1 = S_2$. Thus, neither $S_1$ nor $S_2$ can be the most general solution.

**Definition 15** (generation of equations)
Here is a nondeterministic procedure `gener` that takes as an input a Curry-style term $M$, an enriched first-order context $\Gamma$, and a path $\rho$ (intentionally leading to $M$ in a bigger term) and generates a set of e-equations included in $\mathcal{E}^e_{\Sigma \cup \{c_0, c_1\}}(\Gamma^0, \mathcal{X})$ where $c_0, c_1$ are fresh first-order constants. The procedure follows

1. $\texttt{gener}(x, \Gamma, \rho) = \{\alpha_{x,\rho} \doteq \Gamma(x)\}$ when $x \in \mathrm{Dom}(\Gamma)$ and $\Gamma(x) \neq 0$;
2. $\texttt{gener}(x, \Gamma, \rho) = \{c_0 \doteq c_1\}$ when $x \notin \mathrm{Dom}(\Gamma)$ or $x \in \mathrm{Dom}(\Gamma)$ and $\Gamma(x) = 0$;
3. $\texttt{gener}(MN, \Gamma, \rho) = \{\alpha_{M,\rho\cdot l} \doteq (\forall x : 0)\alpha^0_{MN,\rho}, \alpha^0_{MN,\rho}\langle x := N\rangle \doteq \alpha_{MN,\rho}\} \cup \mathcal{E}_M \cup \mathcal{E}_N$ provided that $N$ is a homogeneous first-order term, $x$ is a fresh first-order variable, $\mathcal{E}_M = \texttt{gener}(M, \Gamma, \rho \cdot l)$ and $\mathcal{E}_N = \texttt{gener}(N, \Gamma, \rho \cdot r)$;
4. $\texttt{gener}(MN, \Gamma, \rho) = \{\alpha_{M,\rho\cdot l} \doteq \alpha_{N,\rho\cdot r} \to \alpha_{MN,\rho}\} \cup \mathcal{E}_M \cup \mathcal{E}_N$ provided that $N$ is not a homogeneous first-order term, $\mathcal{E}_M = \texttt{gener}(M, \Gamma, \rho \cdot l)$ and $\mathcal{E}_N = \texttt{gener}(N, \Gamma, \rho \cdot r)$;
5. nondeterministically choose one of either (5a) or (5b):
   (a) $\texttt{gener}(\lambda x.M, \Gamma, \rho) = \{\alpha_{\lambda x.M,\rho} \doteq \alpha_{x,\rho} \to \alpha_{M,\rho\cdot l}\} \cup \mathcal{E}_M$ where the set of equations $\mathcal{E}_M = \texttt{gener}(M, \Gamma \cup \{x : \alpha_{x,\rho}\}, \rho \cdot l)$,
   (b) $\texttt{gener}(\lambda x.M, \Gamma, \rho) = \{\alpha_{\lambda x.M,\rho} \doteq (\forall x : 0)\alpha_{M,\rho\cdot l}\} \cup \mathcal{E}_M$ where the set of equations $\mathcal{E}_M = \texttt{gener}(M, \Gamma \cup \{x : 0\}, \rho \cdot l)$.

We divide the set of variables $\mathcal{X}$ so that $\mathcal{X} = \mathcal{X}_0 \cup \mathcal{X}_1$ where $\mathcal{X}_0 \cap \mathcal{X}_1 = \emptyset$ and $\mathcal{X}_1 = \{\alpha_{M,\rho} \mid M \in \lambda P,$ and $\rho$ is a path $\}$.

**Theorem 2.** *There exists a nondeterministic algorithm which for each first-order context $\Gamma$ and a Curry style term $M$ has a run that gives a set of e-equations $\mathcal{E}$ such that the following sentences are equivalent:*

1. *There exists a type $\tau$ such that $\Gamma \vdash M : \tau$ has a first-order derivation.*
2. *$\mathcal{E}$ has a solution.*

*Proof.* The algorithm is described by the procedure `gener`. The procedure gives nondeterministically a set of equations $\mathcal{E}$. By straightforward induction on the term $M$, we prove both implications of the theorem. Details are omitted due to lack of space.

### 4.2 Simplification of Equations

Generally, types in equations from the previous subsection contain first-order quantifiers of type 0 and arrows. We get rid of arrows using a procedure similar to the one in Robinson's unification.

In order to shorten the notation, we should denote by $\forall^n\tau$ an e-type of the form $(\forall x_1 : 0)\cdots(\forall x_n : 0)\tau$ where $n \geq 0$. In order to distinguish different $\forall^n$'s, we sometimes supplement them with a subscript.

We present a procedure to simplify equations. The general idea behind the procedure is to unify equations in the fashion of Robinson's unification with additional work connected with pushing explicit substitutions and first-order quantifiers to leaves.

**Definition 16** (simplification procedure)
The procedure processes step by step pairs $(\mathcal{Q}, S)$ where $\mathcal{Q}$ is a sequence of equations to be solved and $S$ is a substitution. The input of the procedure is a pair $(\mathcal{Q}_0, \emptyset)$, where $\mathcal{Q}_0$ is the set of equations we are interested in.

The intended property of the abovementioned substitution $S$ is that if equations in $\mathcal{Q}$ are solvable by a substitution $S'$ then $S' \circ S$ solves $\mathcal{Q}_0$.

The procedure terminates either when it explicitly fails or when the sequence $\mathcal{Q}$ consists only of pairs of one the following three shapes:

$$\forall_1^n \alpha \langle \boldsymbol{x} := \boldsymbol{t} \rangle \doteq \forall_2^n \alpha' \langle \boldsymbol{y} := \boldsymbol{s} \rangle \tag{1}$$

$$\forall_1^n \alpha \langle \boldsymbol{x} := \boldsymbol{t} \rangle \doteq \forall_2^n P(s_1, \ldots, s_m) \tag{2}$$

$$\forall_1^n P(t_1, \ldots, t_n) \doteq \forall_2^n P(s_1, \ldots, s_n) \tag{3}$$

where $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{t}$, $\boldsymbol{s}$ stand for appropriate vectors of variables and terms, and $n \geq 0$.

In the procedure, we use two kinds of type variables: normal variables and travelling variables. Travelling variables are used only in the proof of termination. They may be omitted in a working version of the algorithm. All variables in the input are marked as normal.

At each step, the following cases are checked (we omit cases symmetric wrt. $\doteq$):

1. Let $\mathcal{Q} = \| \ \forall^n (\sigma_1 \to \sigma_2) \langle \boldsymbol{y} := \boldsymbol{t} \rangle \doteq \tau \ \| \ \cdot \mathcal{Q}'$. The present pair is transformed to
$$(\| \ \forall^n (\sigma_1 \langle \boldsymbol{y} := \boldsymbol{t} \rangle \to \sigma_2 \langle \boldsymbol{y} := \boldsymbol{t} \rangle) \doteq \tau \ \| \ \cdot \mathcal{Q}', S).$$

2. Let $\mathcal{Q} = \| \ \forall^n ((\forall x : 0)\sigma_2) \langle \boldsymbol{y} := \boldsymbol{t} \rangle \doteq \tau \ \| \ \cdot \mathcal{Q}'$. The present pair is transformed to
$$(\| \ \forall^n ((\forall x' : 0)\sigma_2 \langle x := x' \rangle \langle \boldsymbol{y'} := \boldsymbol{t'} \rangle) \doteq \tau \ \| \ \cdot \mathcal{Q}', S)$$
where $x'$ does not occur in any of terms in $\boldsymbol{t}$, and $\langle \boldsymbol{y'} := \boldsymbol{t'} \rangle$ are those explicit substitutions for which $y$'s do not occur in $\forall^n$.
3. Let $\mathcal{Q} = \| \ \forall^n P(s_1, \cdots, s_m) \langle \boldsymbol{y} := \boldsymbol{t} \rangle \doteq \tau \ \| \ \cdot \mathcal{Q}'$. The present pair is transformed to
$$(\| \ \forall^n P(s_1 \langle \boldsymbol{y} := \boldsymbol{t} \rangle, \ldots, s_m \langle \boldsymbol{y} := \boldsymbol{t} \rangle) \doteq \tau \ \| \ \cdot \mathcal{Q}', S).$$

4. Let $\mathcal{Q} = \| \ \forall_1^n (\sigma_1 \to \sigma_2) \doteq \forall_2^n (\tau_1 \to \tau_2) \ \| \ \cdot \mathcal{Q}'$. The present pair is transformed to
$$(\| \ \forall_1^n \sigma_1 \doteq \forall_1^n \tau_1 \ \| \cdot \| \ \forall_2^n \sigma_2 \doteq \forall_2^n \tau_2 \ \| \ \cdot \mathcal{Q}', S)$$

5. Let $\mathcal{Q} = \| \ \forall_1^n \alpha \langle \boldsymbol{x} := \boldsymbol{t} \rangle \doteq \forall_2^n \tau_1 \to \tau_2 \ \| \ \cdot \mathcal{Q}'$ where $\boldsymbol{x}$ is the set of variables $x_1, \ldots, x_m$, the vector $\boldsymbol{t}$ is the set of terms $t_1, \ldots, t_m$, and $\alpha$ is a type variable such that no cycle containing $\alpha$ in the graph $G_{\mathcal{Q}}$ has an edge from $E_s$. The present pair is transformed to
$$(\| \ \forall_1^n (\alpha_1 \to \alpha_2) \langle \boldsymbol{x} := \boldsymbol{t} \rangle \doteq \forall_2^n \tau_1 \to \tau_2 \ \| \ \cdot \mathcal{Q}'[\alpha := \alpha_1 \to \alpha_2)], \ [\alpha := \alpha_1 \to \alpha_2] \circ S)$$
where $\alpha_1, \alpha_2$ are fresh variables. Additionally, we mark variables $\alpha_1, \alpha_2$ as travelling.
6. Let $\mathcal{Q} = \| \ \forall_1^n \alpha \langle \boldsymbol{x} := \boldsymbol{t} \rangle \doteq \forall_2^n ((\forall y : 0)\tau) \ \| \ \cdot \mathcal{Q}'$ where $\boldsymbol{x}$ is the set of variables $x_1, \ldots, x_m$, the vector $\boldsymbol{t}$ is the set of terms $t_1, \ldots, t_m$, and $\alpha$ is a type variable

such that no cycle in the graph $G_{\mathcal{Q}}$ contains a vertice with $\alpha$ and an edge from $E_s$ simultaneously. The present pair is transformed to

$$(\parallel \forall_1^n((\forall y' : 0)\alpha_1)\langle \boldsymbol{x} := \boldsymbol{t}\rangle \doteq \forall_2^n((\forall y : 0)\tau) \parallel \cdot \mathcal{Q}'[\alpha := ((\forall y' : 0)\alpha_1)], [\alpha := ((\forall y' : 0)\alpha_1)] \circ S)$$

where $\alpha_1$ is a fresh type variable, and $y'$ is a fresh first-order variable. Additionally, we mark variable $\alpha_1$ as travelling.

7. Let $\mathcal{Q} = \parallel \sigma \doteq \tau \parallel \cdot \mathcal{Q}'$ and let $\sigma \doteq \tau$ be of one of the shapes (1–3). The present pair is transformed to

$$(\mathcal{Q}' \cdot \parallel \sigma \doteq \tau \parallel, S).$$

8. In all other cases fail.

**Theorem 3.** *1. The procedure terminates for all inputs of the form* $(\mathcal{Q}, \emptyset)$.
*2. A system* $\mathcal{Q}$ *has a solution iff the result of the* simplification procedure *applied to* $(\mathcal{Q}, \emptyset)$ *is* $(\mathcal{Q}', S)$ *where* $\mathcal{Q}'$ *has a solution and each equation in the sequence is in one of the forms (1–3) described in Definition 16.*

*Proof.* The termination is obtained due to similar reasoning to the one in Robinson's unification. For the proof of the second claim, one should show that each rule of the *simplification procedure* is sound and complete and then the claim follows by induction. Details are omitted due to lack of space.

### 4.3 Removal of First-Order Quantifiers

We obtain a set of e-equations by means of the *simplification procedure*. These equations have a special form. They still contain first-order quantifiers which do not allow for a direct translation into second-order unification. We introduce a procedure to remove them. The procedure uses as a intermediate data structure a special kind of graph which is defined as follows

**Definition 17** (graph of fixing)
A *graph of fixing* for a set of equations $\mathcal{E}$ is defined as each graph with vertices

$$V_{\mathcal{E}} = X_{\mathcal{E}} \times \mathrm{TV}(\mathcal{E})$$

where $X_{\mathcal{E}}$ is the set of first-order variables quantified in $\mathcal{E}$, and $\mathrm{TV}(\mathcal{E})$ is defined as the set of type variables in $\mathcal{E}$. The edges of such graphs are meant to be unordered pairs.

We will start the procedure of removal of first-order quantifiers with a fixing graph in which an edge between $(x, \alpha)$ and $(x', \alpha')$ informs that there exists an equation with $\alpha$ and $\alpha'$ where $x$ and $x'$ are quantified in the same place in both sides of equation. We will be processing fixing graphs then in order to approach the situation that each edge between $(x, \alpha)$ and $(x', \alpha')$ informs that either $x$ and $x'$ should occur at exactly the same places in terms that result in applying a solution to $\alpha$ and $\alpha'$ respectively.

The procedure that removes quantifiers gives as a result a new set of equations with some negative constraints. These constraints say that some symbol may not occur in a type variable. We remove quantifiers as follows:

**Definition 18** (removal of first-order quantifiers)
The input for the procedure is a triple $(\Sigma, X, \mathcal{E})$ where $\Sigma$ is a signature, $X$ is a set of first-order variables, and $\mathcal{E}$ is a set of e-equations included in $\mathcal{E}^e_\Sigma(X, \mathcal{X})$. The output is a triple $(\Sigma', \mathcal{E}', \phi)$, where $\Sigma'$ is a signature, $\mathcal{E}'$ is a set of e-equations included in $\mathcal{E}^e_{\Sigma'}(\emptyset, \mathcal{X})$, and $\phi : \mathrm{TV}(\mathcal{E}') \to P(X_\mathcal{E})$. Equations are modified according to the following schema:

1. We build a graph of fixing $G_\mathcal{E}$ and $\phi_\mathcal{E}\mathrm{TV}(\mathcal{E}') \to P(X_\mathcal{E})$. They are the result of the hereafter mentioned iteration:
   (a) We begin with $G^0_\mathcal{E} = \langle V_\mathcal{E}, E_0\rangle$ and $\phi_0 : \mathrm{TV}(\mathcal{E}) \to P(X_\mathcal{E})$ where

$$E_0 = \{((x_i, \alpha), (x'_i, \alpha')) \mid [\forall x_1 \ldots \forall x_i \ldots \forall x_n \alpha\langle\cdots\rangle \doteq \forall x'_1 \ldots \forall x'_i \ldots \forall x'_n \alpha'\langle\cdots\rangle] \in \mathcal{E}\}$$
$$\phi_0(\alpha) = \emptyset \text{ for each } \alpha.$$

   (b) We transform $G^n_\mathcal{E}$ into $G^{n+1}_\mathcal{E}$ only if there exists a path $\rho$ in $G^n_\mathcal{E}$ from $(x_i, \alpha)$ to $(x_j, \alpha)$ where $x_i \neq x_j$. We define $E_{n+1}$ and $\phi_{n+1}$ as follows
      i. take an edge in $\rho$ — $((y, \beta), (y', \beta'))$,
      ii. remove the edge — the resulting set is $E_{n+1}$,
      iii. $\phi_{n+1}(\gamma) = \phi_n(\gamma)$ for $\gamma \neq \beta$,
      $\phi_{n+1}(\gamma) = \phi_n(\gamma) \cup \{y\}$ when there is an equation in $\mathcal{E}$

$$\forall z_1 \ldots \forall z_i \ldots \forall z_m \beta\langle \boldsymbol{z}^1 := \boldsymbol{t}\rangle \doteq \forall z'_1 \ldots \forall z'_i \ldots \forall z'_m \beta'\langle\cdots\rangle$$

      where $z_i = y$ and $z'_i = y'$ and $\boldsymbol{z}^1$ does not contain $y$,
      $\phi_{n+1}(\gamma) = \phi_n(\gamma)$ when the former condition does not hold.
2. We produce a new set of equations $\mathcal{E}'$ by means of two steps:
   (a) we generate a function $\psi : V_\mathcal{E} \to \mathrm{Const}$ such that $\psi^{-1}(c)$ is either $\emptyset$ or a connected component in $G_\mathcal{E}$;
   (b) we remove quantifiers in each equation:

$$\forall z_1 \ldots \forall z_i \ldots \forall z_n \alpha\langle \boldsymbol{z}^1 := \boldsymbol{t}^1\rangle \doteq \forall z'_1 \ldots \forall z'_i \ldots \forall z'_n \beta\langle \boldsymbol{z}^2 := \boldsymbol{t}^2\rangle$$

   using the following rules
   – if $(z_i, \alpha)$ and $(z'_i, \beta)$ are in the same connected component then we replace both the variables by $\psi((z_i, \alpha))$;
   – if $(z_i, \alpha)$ and $(z'_i, \beta)$ are in different connected components then
     • if $z_i \in \phi(\alpha)$ we replace $x_i$ and $x_j$ by $\psi((z_i, \beta))$,
     • if $z'_i \in \phi(\beta)$ we replace $x_i$ and $x_j$ by $\psi((z_i, \alpha))$,
     (if both cases hold we take the first option).

The signature we return contains: all the symbols from $\Sigma$, all the symbols from $X$, and all the first-order constants introduced in the abovementioned procedure.

The following fact is necessary to establish the correctnes of the abovementioned definition.

*Property 2.* Let $\forall z_1 \dots \forall z_i \dots \forall z_n \alpha\langle \boldsymbol{z}^1 := \boldsymbol{t}^1\rangle \doteq \forall z'_1 \dots \forall z'_i \dots \forall z'_n \beta\langle \boldsymbol{z}^2 := \boldsymbol{t}^2\rangle$ be an equation. If $G_k$ does not contain the edge $((z_i, \alpha), (z'_i, \beta))$ then either $z_i \in \phi_k(\alpha)$ or $z'_i \in \phi_k(\beta)$.

*Proof.* Induction on $k$.

The following fact explains why we should break paths from $(x_i, \alpha)$ to $(x_j, \alpha)$. Existence of such path means that $x_i$ and $x_j$ are equal.

*Property 3.* Let $\rho$ be a path in $G_k$ for some $k$ and let $S$ be a solution of $\mathcal{E}$. If for each edge $((z_i, \alpha), (z'_i, \beta))$ on $\rho$ and for each equation of the form

$$\forall z_1 \dots \forall z_i \dots \forall z_n \alpha\langle \boldsymbol{z}^1 := \boldsymbol{t}^1\rangle \doteq \forall z'_1 \dots \forall z'_i \dots \forall z'_n \beta\langle \cdots\rangle$$

we have that $z_i \notin \boldsymbol{z}^1$, then there exists a position $w$ such that for each vertice $(y, \gamma)$ in we have that $y$ is on $w$ in $S(\gamma)$.

*Proof.* Induction on the length of $\rho$.

*Property 4.* The procedure of *removal of first-order quantifiers*

1. terminates,
2. $\mathcal{E}$ has a solution $T : \mathcal{X} \rightharpoonup \mathcal{T}^e_\Sigma(X, \emptyset)$ iff the result $\mathcal{E}'$ of *removal of first-order quantifiers* has a solution $T' : \mathcal{X} \rightharpoonup \mathcal{T}^e_{\Sigma'}(X \setminus (\Sigma \cup \mathrm{Dom}(\Gamma)), \emptyset)$

*Proof.* The property (1) is obvious.

The proof of (2) has two parts. We present a scetch of them here.

($\Rightarrow$) If $\mathcal{E}$ has a solution $T$ then we can construct a solution $T'$ of $\mathcal{E}'$. This is done by replacing each constant $x$ in $T(\alpha)$ by $\psi((x, \alpha))$. As there are no paths from $(x_i, \alpha)$ to $(x_j, \alpha)$ in $G_\mathcal{E}$, each first-order variable in $\alpha$ obtains a different constant. Bullets in the point (2b) of Definition 18 guarantee that this operation results in a solution.

($\Leftarrow$) If $\mathcal{E}'$ has a solution $T'$ then we can reconstruct $T$ that solves $E$ by simply replacing fresh constants by first-order variables they replaced. The existence of $\rho$ in the point (1b) of Definition 18 guarantees that only one first-order variable may correspond to a constant in a type variable.

### 4.4 From Equations to Second-Order

We finally obtained a set of e-equations that can easily be transformed to a special form of second-order unification equations. We have to deal with constraints, though. The translation is defined as

**Definition 19** (translation to second-order)

For each type variable $\alpha$, let $A_\alpha$ be the set of all the variables $x$ such that there exists a type $\alpha\langle \boldsymbol{y} := \boldsymbol{t}\rangle\langle x := s\rangle\langle \boldsymbol{z} := \boldsymbol{u}\rangle$ in the set being translated. Assuming $A_\alpha = \{x_1, \dots, x_n\}$, we replace each $\alpha\langle \boldsymbol{y} := \boldsymbol{t}\rangle$ by $F_\alpha(t'_1, \dots, t'_n\}$ where $t'_i = t_j[x_{j+1} := t_{j+1}] \cdots [x_n := t_n]$ if $x_i = y_j$ and $t'_i = x_i$ if there is no $y_j = x_i$.

Constraints are translated to second-order constraints by replacing $\alpha$'s by corresponding $F$'s.

Immediately, we obtain the following property:

*Property 5.* For a given set $\mathcal{E}$ of equations of the form (1–3) in Definition 16, there exists a set $\mathcal{E}'$ of second-order unification equations such that $\mathcal{E}$ is solvable if and only if $\mathcal{E}'$ is solvable.

Moreover, the translation from $\mathcal{E}$ to $\mathcal{E}'$ is effective.

The transformation that allows to get rid of constraints looks as follows

**Definition 20** (removing constraints)
Let $\mathcal{E}$ be a set of second-order equations with constraints $\phi$. For each constant $c \in \Sigma$ we introduce two constants $c_1$ and $c_2$. For each second-order variable $F$ of arity $n$ we introduce two variables $F_1$ and $F_2$ both of arities $n+k$ where $k$ is the number of constants in $\Sigma$. We define two operations $|\cdot|_1$ and $|\cdot|_2$ as follows:

- $|c|_i = c_i$,
- $|f(t_1, \ldots, t_n)|_i = f(|t_1|_i, \ldots, |t_n|_i)$,
- $|F(t_1, \ldots, t_n)|_i = F_i(|t_1|_i, \ldots, |t_n|_i, c_i^1, \ldots, c_i^k)$ where $\{c^1, \ldots, c^k\}$ is the set of all constants in $\Sigma$.

Each equation $t_1 = t_2$ in $\mathcal{E}$ is replaced by a pair of equations $|t_1|_1 = |t_2|_1$ and $|t_1|_2 = |t_2|_2$. For each variable $F$ in $\mathcal{E}$ we supply additional equations $F_1(a, \ldots, a, c_1^1, \ldots, c_1^k) = F_2(a, \ldots, a, c_1^1, \ldots, c_1^k)$ and $F_1(a, \ldots, a, c_2^1, \ldots, c_2^k) = F_2(a, \ldots, a, c_2^1, \ldots, c_2^k)$ where $a$ is a fresh constant. At last for each constraint $\phi(F)$ we supply the equation

$$F_1(a, \ldots, a, c_1^1, \ldots, c_1^k) = F_2(a, \ldots, a, d^1, \ldots, d^k)$$

where $d^i = c_1^i$ if $c^i \notin \phi(F)$ and $d^i = c_2^i$ if $c^i \in \phi(F)$.

Immediately we obtain the following property:

*Property 6.* For a given set $\mathcal{E}$ of equations with constraints $\phi$, there exists a set $\mathcal{E}'$ of second-order unification equations such that $\mathcal{E}$ is solvable if and only if $\mathcal{E}'$ is solvable.

The translation from $\mathcal{E}$ to $\mathcal{E}'$ is effective and involves only equations of the form (1–3) in Definition 21.

### 4.5 Solving of Final Equations

In Subsection 4.2, we obtained sets of equations. Each equation is in one of three forms.

**Definition 21** (head equations)
The sets of equations in one of the following forms

1. $F_1(t_1, \ldots, t_n) = F_2(s_1, \ldots, s_m)$;
2. $F_1(t_1, \ldots, t_n) = P(s_1, \ldots, s_m)$;
3. $P_1(t_1, \ldots, t_n) = P_2(s_1, \ldots, s_m)$;

where $F_1, F_2$ are second-order variables, and $P_1, P_2$ symbols of first-order constants, is called the set of *head equations.*

Now, we describe a procedure to solve such sets of equations.
We need the following facts:

**Theorem 4.** *complete set of solutions for second-order matching For each set $E$ of second-order matching equations, if the set is solvable then it has a finite number of solutions with domains equal to* $\mathrm{TV}(E)$ *and all of them can be effectively generated.*

*Property 7.* If the set $E$ of second-order unification has only equations of the form $F_1(t_1, \ldots, t_n) = F_2(s_1, \ldots, t_m)$, then it has a ground solution provided that the signature has at least one constant symbol.

The first one is proved in [HL78]. The second is obvious — the solution assigns the same term with no arguments on all the second-order variables. This construction is applicable, though, only if we have at least one constant symbol in the signature.

**Definition 22** (solving procedure)
The nondeterministic procedure to solve our second-order equations is defined as follows:

1. Check if there are equations of the form $P_1(t_1, \ldots, t_n) = P_2(s_1, \ldots, s_m)$, where the sides of the equation are different. If so fail else remove all equations of the form $P(t_1, \ldots, t_n) = P(t_1, \ldots, t_n)$
2. Find the complete set $A$ of solutions for all the equations of the form $F(t_1, \ldots, t_n) = P(s_1, \ldots, t_m)$ (the set exists by Theorem 4). If there are no such equations then go to step 5.
3. Choose one of the solutions and apply to all equations.
4. Go to step 1.
5. There are only equations of the form $F_1(t_1, \ldots, t_n) = F_2(s_1, \ldots, t_m)$. These equations have always a solution (provided that there is at least one constant in the signature) so we accept in this case.

We have the following theorem:

**Theorem 5.** *The problem, if a given set of head equations is solvable, is decidable.*

At last, we obtain:

**Theorem 6.** *The first-order type inference problem is decidable.*

*Proof.* A consequence of Theorem 2, Theorem 3, Fact 5 and Theorem 5.

## 5 Acknowledgements

I would like to thank Paweł Urzyczyn for encouraging me to work on the problem. He is also a person to thank for his digging in early versions of the paper. His remarks were very helpful during the preparation of the document. Also, I would

like to thank Gilles Dowek for a discussion we had about the problems. In fact, much of the ideas in the proof presented in Subsection 4.5 is due to him. The present proof, due to his remarks, is much simpler than my primary version. Anonymous referees contributed to better presentation of the paper so my thanks go for them too.

## References

[Dow93] Gilles Dowek, *The undecidability of typability in the lambda-pi-calculus*, Typed Lambda Calculi and Applications (M. Bezem and J.F. de Groote, eds.), LNCS, no. 664, 1993, pp. 139–145.

[Gol81] W. D. Goldfarb, *The undecidability of the second-order unification problem*, TCS (1981), no. 13, 225–230.

[HHP87] R. Harper, F. Honsell, and G. Plotkin, *A Framework for Defining Logics*, Proceedings of Logic in Computer Science, 1987, pp. 194–204.

[HL78] G. Huet and B. Lang, *Proving and applying program transformations expressed with second order patterns*, Acta Informatica (1978), no. 11, 31–55.

[Pfe91] Frank Pfenning, *Logic Programming in the LF Logical Framework*, Logical Frameworks (Gherard Huet and Gordon Plotkin, eds.), Cambridge University Press, 1991.

[Sch98] A. Schubert, *Second-order unification and type inference for church-style polymorphism*, Proc. of POPL, 1998.

[SU98] M.H. Sørensen and P. Urzyczyn, *Lectures on Curry-Howard Isomorphism*, Tech. Report 14, DIKU, 1998.

[TvD88] A.S. Troelstra and D. van Dalen, *Constructivism in Mathematics, An Introduction, Volume II*, Studies in Logic and the Foundations of Mathematics, vol. 123, North-Holland, 1988.

[vBLRU97] Stefan van Bakel, Luigi Liquori, Simona Ronchi Della Rocca, and PawełUrzyczyn, *Comparing cubes of typed and type assignment systems*, Annals of Pure and Applied Logic (1997), no. 86, 267–303.

[Vor96] Andrei Voronkov, *Proof search in intuitionistic logic based on constraint satisfaction*, Theorem Proving with Analytic Tableaux and Related Methods (Terrasini, Palermo) (P. Miglioli, U. Moscato, D. Mundici, and M. Ornaghi, eds.), LNAI, no. 1071, Springer Verlag, 1996, pp. 312–329.