Currying Second-Order Unification Problems*

Jordi Levy[1] and Mateu Villaret[2]

[1] IIIA, CSIC, Campus de la UAB, Barcelona, Spain.
http://www.iiia.csic.es/~levy
[2] IMA, UdG, Campus de Montilivi, Girona, Spain.
http://www.ima.udg.es/~villaret

Abstract. The Curry form of a term, like $f(a, b)$, allows us to write it, using just a single binary function symbol, as $@(@(f, a), b)$. Using this technique we prove that the signature is not relevant in second-order unification, and conclude that one binary symbol is enough.
By currying variable applications, like $X(a)$, as $@(X, a)$, we can transform second-order terms into first-order terms, but we have to add beta-reduction as a theory. This is roughly what it is done in explicit unification. We prove that by currying only constant applications we can reduce second-order unification to second-order unification with just one binary function symbol. Both problems are already known to be undecidable, but applying the same idea to context unification, for which decidability is still unknown, we reduce the problem to context unification with just one binary function symbol.
We also discuss about the difficulties of applying the same ideas to third or higher order unification.

1 Introduction

The Curry form of a term, like $f(a, b)$, allows us to write it, using just a single binary symbol, as $@(@(f, a), b)$, where $@$ denotes the explicit application. This helps to solve unification problems. In first-order logic, this transformation reduces a unification problem to a new unification problem containing a single binary symbol. The size of the new problem [and of the unifier] is similar to the size of the original problem [and of the original unifier]. So, from the point of view of complexity there is not a significant difference, but in practical implementations this allows representing terms as binary trees, and contexts as subterms, and has been used in term indexing data structures [GNN01].

In second-order logic the transformation is not so obvious. We can currify constant symbol applications and second-order variable applications, obtaining a first-order term. For instance, for $f(X(a), Y)$, where X is a second-order variable, we obtain $@(@(f, @(X, a)), Y)$, where both X and Y are now first-order variables. However, solvability of unification problems is not preserved by such transformation, unless we consider some form of first-order unification modulo

* This research has been partially supported by the CICYT Research Projects DENOC (BFM2000-1054-C02), LOGFAC, and TIC2001-2392-C03-01

β-reduction for solving the new problem. For instance, the second-order unification problem $F(G(a), b) \stackrel{?}{=} g(a)$ is solvable, whereas its first-order Curry form $@(@(F, @(G, a)), b) \stackrel{?}{=} @(g, a)$ is unsolvable. Moreover, the right-hand side of the β-equation $(\lambda x \,.\, t_1)t_2 = t_1[t_2/x]$ is a meta-term, unless we make substitution explicit [ACCL98]. Roughly speaking this is what is done in the so called *explicit unification* [DHK00,BM00].

Here, we propose to currify function symbol applications, but not variable applications. Therefore, the new problem we get is also a second-order unification problem. For instance, for $F(G(a), b) \stackrel{?}{=} g(a)$, we get $F(G(a), b) \stackrel{?}{=} @(g, a)$, that is also solvable. In this case, we do not reduce the order of the unification problem, but we reduce the number of function symbols to just one: the application symbol @. It can be argued that this reduction is useless, since second-order unification [Gol81] was already known to be undecidable for just one binary function symbol [Far91], although applying the reduction to the results of [LV00] proves that second-order unification is undecidable for one binary function symbol and one second-order variable occurring four times. Moreover, the same reduction is applicable to context unification [Com98], for which decidability is still unknown [Com98,LV01,SS96,Lev96,SS98,SSS98,SSS99], and it allows concentrating the efforts in a very simple signature. We also think that currying could help to simplify the signature used in higher-order matching, and this could help to prove its decidability (or undecidability).

If we currify function applications in a second-order [or context] unification problem, it is easy to prove that, if the original problem is solvable, then its Curry form is also solvable: we can currify the unifier of the original problem to obtain a unifier of its Curry form. However, the converse is not true and, in general, solvability is not preserved by currying, as the following examples prove.

Example 1. The following context unification problem

$$\begin{array}{l} \quad g(\ F(G(a)),\, F(a),\quad G(a)\quad\ ) \stackrel{?}{=} \\ \stackrel{?}{=} g(\ f(a,b),\quad H(a,b),\, H(X,a)\ ) \end{array}$$

is unsolvable. However, its Curry form

$$\begin{array}{l} \quad @(@(@(g,\, F(G(a))\quad\ ),\, F(a)\quad),\, G(a)\quad\ ) \stackrel{?}{=} \\ \stackrel{?}{=} @(@(@(g,\, @(@(f,a),b)\ ),\, H(a,b)\ ),\, H(X,a)\ ) \end{array}$$

is solvable and has the following unifier

$$\begin{array}{l} \sigma(F) = \lambda x \,.\, @(x, b) \\ \sigma(G) = \lambda x \,.\, @(f, x) \\ \sigma(H) = \lambda x \,.\, \lambda y \,.\, @(x, y) \\ \sigma(X) = f \end{array}$$

Similarly, the following second-order unification problem

$$\begin{array}{l} \quad g(\ F(G(a)),\, F(G(a')),\, F(a),\quad F(a'),\quad G(a),\quad G(a')\quad\ ) \stackrel{?}{=} \\ \stackrel{?}{=} g(\ f(a,b),\quad f(a',b),\quad H(a,b),\, H(a',b),\, H(X,a),\, H(X,a')\ ) \end{array}$$

is also unsolvable, whereas its Curry form is solvable.

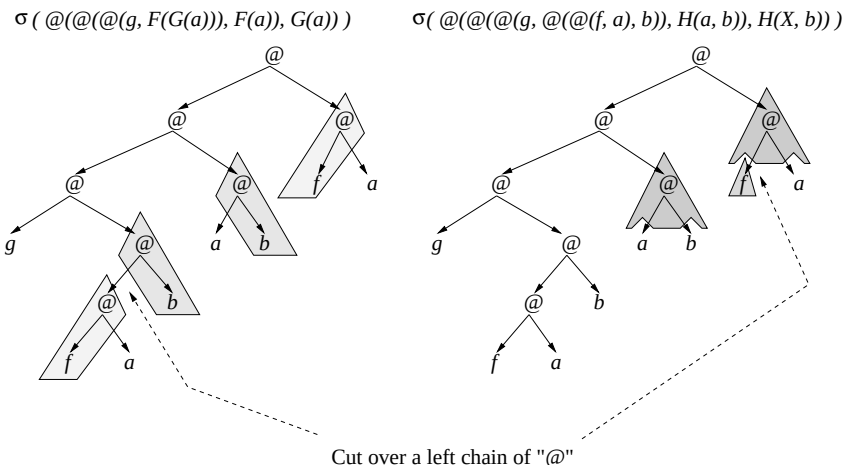


Fig. 1. Graphic representation of the unifier of curried context unification problem of Example 1.

In the previous example, $\sigma(F)$, $\sigma(G)$, $\sigma(H)$ and $\sigma(X)$ are not "well-typed", i.e. they are not the Curry form of any well-typed term. For instance, $\sigma(F) = \lambda x\,.\,@(x, b)$ is the Curry form of $\lambda x\,.\,x(b)$, but this term is third-order typed (and F is a second-order typed variable), and $\sigma(G) = \lambda x\,.\,@(f, x)$ is the Curry form of $\lambda x\,.\,f(x)$, but f has two arguments. This disallows us to reconstruct a unifier for the original unification problem from the unifier we get for its Curry form.

We can also see that the original unification problems contain variables that "touch". For instance, F touches G in $F(G(a))$, and H touches X in $H(X, a)$. We will prove, for second-order and for context unification, that, if no variable touches any other variable, then solvability of the problems is preserved in both directions by our partial currying. It is easy to reduce second-order and context unification problems to problems accomplishing such property. Therefore, we conclude that second-order and context unification can be both reduced to the partial Curry form, where only a binary function symbol @ is used.

It is well known that word unification [Mak77] is an special case of context unification. Plandowski [Pla99] proves that if σ is a most general unifier of a word unification problem $t \stackrel{?}{=} u$, then any substring of $\sigma(t)$ "is over a cut", i.e. there exists an occurrence of the substring in $\sigma(t)$ that is not completely inside the instance of a variable. Something similar can be proved for second-order and for context unification. The pathology of Example 1 is due to the existence of a cut over a left chain of @ ended by a constant. For instance, in the example, the left chain $@(@(f, \ldots), \ldots)$ is "cut" by $F(G(\ldots))$, i.e. one piece is inside $\sigma(F)$ and another inside $\sigma(G)$ (see Figure 1). If variables "do not touch" this situation is avoided, and satisfiability is preserved. Our main result could be proved using a version of Plandowski's theorem for second-order unification, but the proof would be longer than the one we present in this paper.

This paper proceeds as follows. In Section 2 we introduce some standard definitions and results about second-order and context unification. Most of our results hold for second-order and for context unification, and sometimes we do not make the distinction explicit. In Section 3 we define the partial Curry forms where only function symbols applications are made explicit. In Section 4 we define a labeling on Curry forms that is used to characterize "well-typed" terms, i.e. terms that are the Curry form of some well-built term. In Section 5 we prove our main result: second-order and context unification can be reduced to a simplified form where only a single binary function symbol and constants are used. We conclude in Section 6 with a discussion about the difficulties to extended these results to higher order.

2 Preliminary Definitions

A *second-order signature* Σ is a finite disjoint union of finite sets of symbols $\Sigma = \bigcup_{n\geq 0} \Sigma_n$, where symbols $f \in \Sigma_n$ are said to be n-ary, noted $\text{arity}(f) = n$. We distinguish between *constant symbols*, when $\text{arity}(a) = 0$, and *function symbols*, when $\text{arity}(f) > 0$. Similarly, we define the set of variables $\mathcal{X} = \bigcup_{n\geq 0} \mathcal{X}_n$, and distinguish between first-order variables (the set $\mathcal{X}_0$) and second-order variables (the rest of variables $\bigcup_{n\geq 1} \mathcal{X}_n$). We use lambda bindings and the usual notion of bound and free variables. For simplicity, we assume that bound and free variables have distinct names, and use lower case letters for the bound variables and upper case letters for free variables.

The set of terms $\mathcal{T}(\Sigma, \mathcal{X})$ is defined as in [Gol81]. A first-order term is either a constant symbol $a \in \Sigma_0$, a first-order variable $X \in \mathcal{X}_0$, or has the form $f(t_1, \ldots, t_n)$ or $X(t_1, \ldots, t_n)$, where $f \in \Sigma_n$, $X \in \mathcal{X}_n$, and t_i's are first-order terms. A second-order term has the form $\lambda x_1 \ldots \lambda x_n \,.\, t$, where t is a first-order term, $n \geq 1$, and x_i's are (bound) first-order variables. In other words, we assume that any term is written in $\beta\eta$-long normal form, and we do not consider constants of third or higher order. Therefore, λ-abstractions always appear in the head of the term. The arity of a term is the number of λ-abstractions that it has in the head. Therefore, first-order terms are the terms of arity zero.

A *second-order unification problem* is a pair of first-order terms, noted $t \stackrel{?}{=} u$. Notice that all variables of $t \stackrel{?}{=} u$ occur free.

A *second-order substitution* σ is a set of (variable,term) pairs, like $[X_1 \mapsto t_1] \ldots [X_n \mapsto t_n]$, where X_i and t_i have the same arity. Therefore, instances of second-order variables contain λ-abstractions. The application of a substitution σ to a first-order term t is defined recursively as follows

$$\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$$

$$\begin{array}{ll}
\sigma(X) = t & \text{if the pair } X \mapsto t \text{ is in } \sigma \\
\sigma(X) = X & \text{otherwise} \\
\\
\sigma(F(t_1, \ldots, t_n)) = \rho(u) & \text{if the pair } F \mapsto \lambda x_1 \cdots \lambda x_n \,.\, u \text{ is in } \sigma, \text{ and} \\
 & \text{where } \rho = [x_1 \mapsto \sigma(t_1)] \ldots [x_n \mapsto \sigma(t_n)] \\
\sigma(F(t_1, \ldots, t_n)) = F(\sigma(t_1), \ldots, \sigma(t_n)) & \text{otherwise}
\end{array}$$

Notice that in the previous definition we avoid the definition of instances of a second-order terms, and therefore all problems related with the variable capture.

A substitution σ is said to be a *unifier* (or solution) of a second-order unification problem $t \stackrel{?}{=} u$ if $\sigma(t) = \sigma(u)$. The definition of *most general unifier* is standard.

A *context unification problem* is also a pair of first-order terms $t \stackrel{?}{=} u$ over a second-order signature. In the ambit of context unification, second-order variables are called *context variables*. A *context substitution* is a second-order substitution $[X_1 \mapsto t_1] \dots [X_n \mapsto t_n]$ where, for all second-order term $t_i = \lambda x_1 \cdots \lambda x_n \,.\, u$, every x_j occurs exactly once in u. Then, a *context unifier* of a context unification problem $t \stackrel{?}{=} u$ is a context substitution σ satisfying $\sigma(t) = \sigma(u)$. Notice that second-order and context unification problems have the same presentation, and any solvable context unification problem is also solvable viewed as a second-order unification problem.

Sometimes, context unification is defined restricting context variables to be unary. Here we consider n-ary variables, and use bound variables to denote the "holes" of the context (instead of the box $\Box$ used by other authors).

If nothing is said, the signature of a problem is given by the set of constants that it contains and a denumerable infinite set of variables, for every arity. For technical reasons we also assume that the signature contains, at least, a binary function symbol and a constant (that can be added if the problem does not contain any). The following is a basic property of most general second-order [and context] unifiers that will be required in some proofs. It ensures that the signature does not play an important role w.r.t. the decidability of the problem.

Property 1. Let $t \stackrel{?}{=} u$ be a second-order or a context unification problem, and σ be a most general unifier. Then, for any variable X, $\sigma(X)$ does not contain constants not occurring in the problem $t \stackrel{?}{=} u$.

Proof: Suppose that a most general unifier σ introduces a constant f not occurring in the problem. Then we can replace everywhere this constant by a fresh variable F of the same arity and get another unifier that is more general than σ (we can instantiate F by $\lambda x_1 \dots \lambda x_n \,.\, f(x_1, \dots, x_n)$, but not vice versa). This contradicts the fact that σ is most general. ■

3 Currying Terms

Definition 1. *Given a signature* $\Sigma = \bigcup_{n \geq 0} \Sigma_n$*, the* curried signature $\Sigma^c = \bigcup_{n \geq 0} \Sigma^c_n$ *is defined by*

$$\begin{array}{ll} \Sigma^c_0 = \bigcup_{n \geq 0} \Sigma_n & \\ \Sigma^c_2 = \{@\} & \\ \Sigma^c_n = \emptyset & \text{for } n \neq 0, 2 \end{array}$$

The currying function $\mathcal{C} : \mathcal{T}(\Sigma, \mathcal{X}) \to \mathcal{T}(\Sigma^c, \mathcal{X})$ *is defined recursively as follows:*

$$\begin{aligned}
&\mathcal{C}(a) = a \\
&\mathcal{C}(x) = x \\
&\mathcal{C}(f(t_1, \ldots, t_n)) = @(\overset{n}{\cdots} @(f, \mathcal{C}(t_1)) \overset{n}{\cdots}, \mathcal{C}(t_n)) \\
&\mathcal{C}(F(t_1, \ldots, t_n)) = F(\mathcal{C}(t_1), \ldots, \mathcal{C}(t_n)) \\
&\mathcal{C}(\lambda x \,.\, t) = \lambda x \,.\, \mathcal{C}(t)
\end{aligned}$$

for any constant $a \in \Sigma_0$*, bound variable* x*, function symbol* $f \in \Sigma_n$*, and variable* $F \in \mathcal{X}_n$*.*

The currying function is injective, but it is not onto, as the following definition suggests.

Definition 2. *Given a term* $t \in \mathcal{T}(\Sigma^c, \mathcal{X})$*, we say that it is* well-typed *(w.r.t.* Σ*), if* $\mathcal{C}^{-1}(t)$ *is defined, i.e. if there exists a term* $u \in \mathcal{T}(\Sigma, \mathcal{X})$ *such that* $\mathcal{C}(u) = t$*.*

Lemma 1. *If the second-order [context] unification problem* $t \stackrel{?}{=} u$ *over* Σ *is solvable, then the second-order [context] unification problem* $\mathcal{C}(t) \stackrel{?}{=} \mathcal{C}(u)$ *over* Σ^c *is also solvable.*

Proof: Let σ be a unifier of $t \stackrel{?}{=} u$, then it is easy to prove that the substitution $\sigma_{\mathcal{C}}$ defined as $\sigma_{\mathcal{C}}(F) = \mathcal{C}(\sigma(F))$ is a unifier of $\mathcal{C}(t) \stackrel{?}{=} \mathcal{C}(u)$. ■

In fact, we have proved a stronger result: given a unifier σ of $t \stackrel{?}{=} u$, we can find a unifier $\sigma_{\mathcal{C}}$ of $\mathcal{C}(t) \stackrel{?}{=} \mathcal{C}(u)$ that satisfies the commutativity property $\mathcal{C}(\sigma(t)) = \sigma_{\mathcal{C}}(\mathcal{C}(t))$. This commutativity property is represented by the following diagram:

$$\begin{array}{ccc}
t \stackrel{?}{=} u & \xrightarrow{\mathcal{C}} & \mathcal{C}(t) \stackrel{?}{=} \mathcal{C}(u) \\
\Big\downarrow \sigma & \Longrightarrow^{\mathcal{C}} & \Big\downarrow \sigma_{\mathcal{C}} \\
\sigma(t) & \xrightarrow{\mathcal{C}} & \sigma_{\mathcal{C}}(\mathcal{C}(t))
\end{array}$$

Unfortunately, as it is shown in Example 1, the converse is not true. Given a unifier of $\mathcal{C}(t) \stackrel{?}{=} \mathcal{C}(u)$ it is not always possible to obtain a unifier of $t \stackrel{?}{=} u$. In the next Section, we describe sufficient conditions to ensure that the inverse construction is possible.

4 Labeling Terms

The first step to find a sufficient condition ensuring that the currying function preserves satisfiability is to characterize well-typed curried terms. This is done by labeling application symbols @ with the "arity" of their left argument, and using a "hat"to mark the roots of right arguments. If left arguments always have positive arity, and right arguments always have arity zero, then the term is well-typed.

Definition 3. *Given a signature* $\Sigma = \bigcup_{n\geq 0} \Sigma_n$*, the* labeled signature $\Sigma^L = \bigcup_{n\geq 0} \Sigma^L_n$ *is defined by:*

$$\begin{array}{ll} \Sigma^L_0 = \bigcup_{n\geq 0} \Sigma_n & \\ \Sigma^L_2 = \{@^l, \widehat{@^l} \mid l \in \{\ldots, -1, 0, 1, \ldots\}\} & \\ \Sigma^L_n = \emptyset & \text{for } n \neq 0, 2 \end{array}$$

The labeling functions $\mathcal{L}, \widehat{\mathcal{L}} : \mathcal{T}(\Sigma^c, \mathcal{X}) \to \mathcal{T}(\Sigma^L, \mathcal{X})$ *are defined by the following rules:*

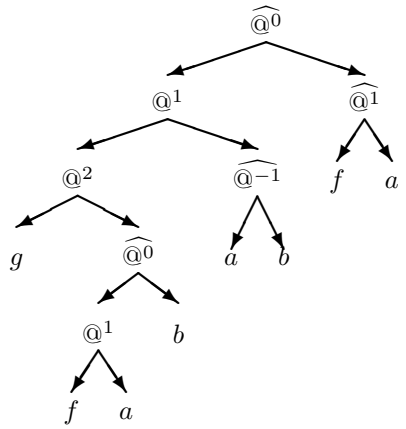
1. *If the left child of an* @ *is an* n*-ary symbol* $f \in \Sigma_n$*, then it has label* $l = \text{arity}(f) - 1 = n - 1$*.*
2. *If the left child of an* @ *is a variable* $X \in \mathcal{X}$*, or a bound variable, then it has label* -1*, regardless the arity of the variable is.*
3. *If the left child of an* @ *is another* @ *with label* n*, then it has label* $n - 1$*.*

In the case of $\widehat{\mathcal{L}}$ *we also use the following rule:*

4. *If an* @ *is the right child of another* @*, or it is the child of a variable, or it is the root of the term, then, apart from the label, it also has a hat.*

Example 2. The $\widehat{\mathcal{L}}$-labeling of the term $\sigma(\mathcal{C}(t))$, used in Example 1 and shown in Figure 1, is as follows.

$$\widehat{@^0}(@^1(@^2(g, \widehat{@^0}(@^1(f, a), b)), \widehat{@^{-1}}(a, b)), \widehat{@^1}(f, a))$$

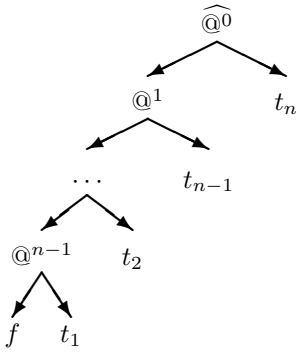


Notice that labels can be negative numbers. These negative labels do not appear in labellings of "well-typed" terms.

Based on these labels, it is easy to characterize well-typed terms.

Lemma 2. *A term* $t \in \mathcal{T}(\Sigma^c, \mathcal{X})$ *is well-typed if, and only if,* $\widehat{\mathcal{L}}(t)$ *does not contain application symbols with negative labels (*$@^{-n}$*, for* $n > 0$*) or with hat and non-zero labels (*$\widehat{@^n}$ *with* $n \neq 0$*).*

Proof: The only if implication is obvious. For the if implication, assume that the labeling $\widehat{\mathcal{L}}(t)$ does not contain $@^{-n}$, with $n > 0$, or $\widehat{@^n}$, with $n \neq 0$. Then, any @ symbol is in a sequence of the form:



where the node $\widehat{@^0}$ is a right child of another @, or the child of a variable F, or the root of the term. We can prove that this is the currying of $f(\mathcal{C}^{-1}(t_1), \ldots, \mathcal{C}^{-1}(t_n))$ that is a well constructed term, because f has n arguments and arity n. ■

5 When Variables do not Touch

In this section, we try to find sufficient conditions ensuring that, when we have a unifier for $\mathcal{C}(t) \stackrel{?}{=} \mathcal{C}(u)$, we can find a unifier for $t \stackrel{?}{=} u$. The strategy to prove this result is summarized in the following diagram:

$$\begin{array}{ccccc}
t \stackrel{?}{=} u & \xrightarrow{\mathcal{C}} & \mathcal{C}(t) \stackrel{?}{=} \mathcal{C}(u) & \xrightarrow{\widehat{\mathcal{L}}} & \widehat{\mathcal{L}}(\mathcal{C}(t)) \stackrel{?}{=} \widehat{\mathcal{L}}(\mathcal{C}(u)) \\
\downarrow \sigma & \xleftarrow{\mathcal{C}^{-1}} & \downarrow \sigma_{\mathcal{C}} & \xRightarrow{\widehat{\mathcal{L}}} & \downarrow \sigma_{\widehat{\mathcal{L}}} \\
\sigma(t) & \xleftarrow{\mathcal{C}^{-1}} & \sigma_{\mathcal{C}}(\mathcal{C}(t)) & \xrightarrow{\widehat{\mathcal{L}}} & \sigma_{\widehat{\mathcal{L}}}(\widehat{\mathcal{L}}(\mathcal{C}(t)))
\end{array}$$

We will find a condition that makes the right square commute (Lemma 5). Then we will prove that when the right square commutes, then the left one also commutes (Lemma 6). This second commutativity property ensures that the currying transformation preserves satisfiability.

The sufficient condition we have found is based on the following definition.

Definition 4. *Given a term $t \in \mathcal{T}(\Sigma, \mathcal{X})$, we say that two variables $X, Y \in \mathcal{X}$* touch*, if t contains a subterm of the form $X(t_1, \ldots, Y(u_1, \ldots, u_m), \ldots, t_n)$.*

In the context unification problem of Example 1, the variable F touches G, and the variable H touches X.

For technical reasons, before proving that the first square commutes (Lemma 5) we prove the same result using a variant of the labeling function where hats are not considered (notice that in Lemma 3 $\mathcal{L}$'s have no hats).

Lemma 3. *If the variables of $t \stackrel{?}{=} u$ do not touch, and $\sigma_{\mathcal{C}}$ is a most general unifier of $\mathcal{C}(t) \stackrel{?}{=} \mathcal{C}(u)$, then the substitution $\sigma_{\mathcal{L}}$ defined by*

$$\sigma_{\mathcal{L}}(F) = \mathcal{L}(\sigma_{\mathcal{C}}(F))$$

is a most general unifier of $\mathcal{L}(\mathcal{C}(t)) \stackrel{?}{=} \mathcal{L}(\mathcal{C}(u))$, and satisfies

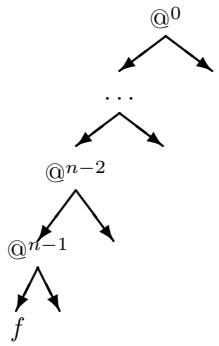
$$\sigma_{\mathcal{L}}(\mathcal{L}(\mathcal{C}(t))) = \mathcal{L}(\sigma_{\mathcal{C}}(\mathcal{C}(t)))$$

Proof: First, we prove that

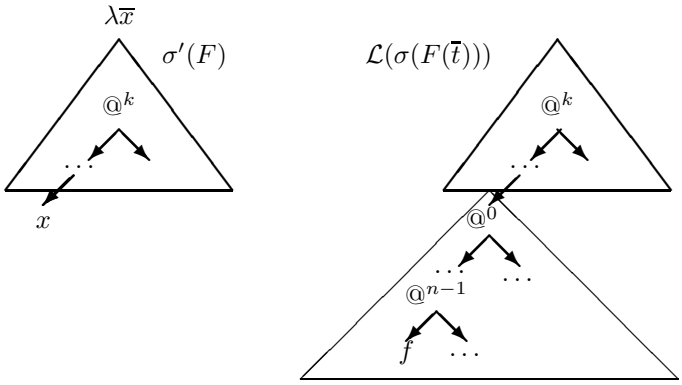
$$\sigma_{\mathcal{L}}(\mathcal{L}(\mathcal{C}(t))) = \mathcal{L}(\sigma_{\mathcal{C}}(\mathcal{C}(t)))$$

As far as $\sigma_{\mathcal{C}}$ and $\sigma_{\mathcal{L}}$ only differ in the introduction of labels, both terms have the same form, except for the labels. Therefore, we only have to compare the labels of the corresponding @'s in both terms. There are two cases:

- If the occurrence of the @ is outside the instance of any variable, then this @ already occurs in $\mathcal{C}(t)$, and it is in a sequence of the form:



  where the f and all the @'s in between already occur in $\mathcal{C}(t)$ (they have not been introduced by an instantiation either). Thus, the @ gets the same label in $\sigma_{\mathcal{L}}(\mathcal{L}(\mathcal{C}(t))$ as in $\mathcal{L}(\sigma_{\mathcal{C}}(\mathcal{C}(t)))$, because this label only depends on the left descendants, and they have not been introduced by $\sigma_{\mathcal{C}}$ or $\sigma_{\mathcal{L}}$.
- If the @ is inside the instance of a variable F, we have to prove that it gets the same label in $\sigma_{\mathcal{L}}(\mathcal{L}(F(t_1, \ldots, t_n))) = \sigma_{\mathcal{L}}(F)(\sigma_{\mathcal{L}}(\mathcal{L}(t_1)), \ldots, \sigma_{\mathcal{L}}(\mathcal{L}(t_1)))$ as in $\mathcal{L}(\sigma_{\mathcal{C}}(F(t_1, \ldots, t_n)))$. In the first case we label $\sigma_{\mathcal{C}}(F)$ before instantiating (so we have bound variables in the place of the arguments), whereas in the second case we label $\sigma_{\mathcal{C}}(F)$ after instantiating (so we already have the arguments t_i). As we will see, in both cases the labels we get are the same. The root of one of the arguments t_i can be a left descendant of the @, and its label will depend on such argument. However, if variables do not touch, the head of any argument t_i of F is a constant, and the head of $\mathcal{C}(t_i)$ is either a 0-ary constant a or an @ with label 0. Therefore, the labels of the ancestors of the argument inside $\sigma_{\mathcal{C}}(F)$ will be the same if we replace the argument by a bound-variable, and the label of the corresponding @ inside $\sigma_{\mathcal{L}}(F)$ will be the same.



Similarly, we can prove $\sigma_{\mathcal{L}}(\mathcal{L}(\mathcal{C}(u))) = \mathcal{L}(\sigma_{\mathcal{C}}(\mathcal{C}(u)))$. As $\sigma_{\mathcal{C}}(\mathcal{C}(t)) = \sigma_{\mathcal{C}}(\mathcal{C}(u))$, we can conclude that $\sigma_{\mathcal{L}}$ is a unifier of $\mathcal{L}(\mathcal{C}(t)) \stackrel{?}{=} \mathcal{L}(\mathcal{C}(u))$.

Given a unifier of $\mathcal{L}(\mathcal{C}(t)) \stackrel{?}{=} \mathcal{L}(\mathcal{C}(u))$, we can find a unifier of $\mathcal{C}(t) \stackrel{?}{=} \mathcal{C}(u)$ by removing labels. Using this idea, it is easy to prove that, if $\sigma_{\mathcal{C}}$ is most general for $\mathcal{C}(t) \stackrel{?}{=} \mathcal{C}(u)$, then $\sigma_{\mathcal{L}}$ is also most general for $\mathcal{L}(\mathcal{C}(t)) \stackrel{?}{=} \mathcal{L}(\mathcal{C}(u))$. Otherwise, there would be a unifier more general than $\sigma_{\mathcal{L}}$, and removing labels we could obtain a unifier more general than $\sigma_{\mathcal{C}}$. ■

The following is a technical lemma that we need in the proof of Lemma 5.

Lemma 4. *If the variables of $t \stackrel{?}{=} u$ do not touch, and $\sigma_{\mathcal{C}}$ is a most general unifier of $\mathcal{C}(t) \stackrel{?}{=} \mathcal{C}(u)$, then the arguments t_i of any variable F never occur as left child of an @ in $\sigma_{\mathcal{C}}(\mathcal{C}(t))$.*

Proof: As $\mathcal{C}(t)$ and $\mathcal{C}(u)$ are trivially well-typed, by Lemma 2, $\mathcal{L}(\mathcal{C}(t))$ and $\mathcal{L}(\mathcal{C}(u))$ will not contain @'s with negative labels. Let $\sigma_{\mathcal{L}}$ be the most general unifier of $\mathcal{L}(\mathcal{C}(t)) \stackrel{?}{=} \mathcal{L}(\mathcal{C}(u))$ given by Lemma 3. Now, by Property 1, as $\sigma_{\mathcal{L}}$ is a most general unifier, for any variable F, $\sigma_{\mathcal{L}}(F)$ will not contain @'s with negative labels, either. We can conclude then that the head of any argument t_i of F can not be a left child of an @. As far as the heads of $\sigma_{\mathcal{L}}(t_i)$ have zero label or are 0-ary constants, this situation would introduce a negative label in some @ inside $\sigma_{\mathcal{L}}(F)$. ■

Lemma 5. *If the variables of $t \stackrel{?}{=} u$ do not touch, and $\sigma_{\mathcal{C}}$ is a most general unifier of $\mathcal{C}(t) \stackrel{?}{=} \mathcal{C}(u)$, then the substitution $\sigma_{\widehat{\mathcal{L}}}$ defined by*

$$\sigma_{\widehat{\mathcal{L}}}(F) = \widehat{\mathcal{L}}(\sigma_{\mathcal{C}}(F))$$

is a most general unifier of $\widehat{\mathcal{L}}(\mathcal{C}(t)) \stackrel{?}{=} \widehat{\mathcal{L}}(\mathcal{C}(u))$, and satisfies

$$\sigma_{\widehat{\mathcal{L}}}(\widehat{\mathcal{L}}(\mathcal{C}(t))) = \widehat{\mathcal{L}}(\sigma_{\mathcal{C}}(\mathcal{C}(t)))$$

Proof: We already know that both terms have the same form and the same labels, thus we only have to prove that they have the same hats. Again, there are two cases:

– If the occurrence of the @ is outside the instance of any variable, then the only situation we have to consider is the following. If the @ has as father a variable F in $\mathcal{C}(t)$, and after instantiation, it becomes a left child of an @ inside $\sigma_{\widehat{\mathcal{L}}}(F)$, then it could loose the hat. However, if variables do not touch, this situation is not possible because, by Lemma 4, arguments t_i of F never occur as left child of an @ in $\sigma_{\widehat{\mathcal{L}}}(F(t_1, \ldots, t_n))$.
– If the occurrence of the @ is inside the instance of a variable F, then we have to prove that the fact that @ has a hat or not, does not depend on the arguments of F. This is obvious because this fact does not depend on the descendants of the @. As in Lemma 3, this allows us to replace arguments by bound variables and get a unifier $\sigma_{\widehat{\mathcal{L}}}$ for our problem.

Using the argument of Lemma 3, we conclude that $\sigma_{\widehat{\mathcal{L}}}$ is a most general unifier of $\widehat{\mathcal{L}}(\mathcal{C}(t)) \stackrel{?}{=} \widehat{\mathcal{L}}(\mathcal{C}(u))$. ■

Lemma 6. *If the variables of $t \stackrel{?}{=} u$ do not touch, and $\sigma_{\mathcal{C}}$ is a most general unifier of $\mathcal{C}(t) \stackrel{?}{=} \mathcal{C}(u)$, then there exists a most general unifier σ of $t \stackrel{?}{=} u$ that satisfies*

$$\mathcal{C}(\sigma(t)) = \sigma_{\mathcal{C}}(\mathcal{C}(t))$$

Proof: Let $\sigma_{\widehat{\mathcal{L}}}$ be the most general unifier of $\widehat{\mathcal{L}}(\mathcal{C}(t)) \stackrel{?}{=} \widehat{\mathcal{L}}(\mathcal{C}(u))$ given by Lemma 5. As $\mathcal{C}(t)$ and $\mathcal{C}(u)$ are well-typed, by Lemma 2, they do not contain negative labels nor hats over non-zero labeled @'s. Then, by Property 1, $\sigma_{\widehat{\mathcal{L}}}$ does not introduce such kind of labels or hats. Therefore, as $\sigma_{\widehat{\mathcal{L}}}(F)$ is defined as the labeling of $\sigma_{\mathcal{C}}(F)$, using again Lemma 2, $\sigma_{\mathcal{C}}(F)$ will be well-typed, and we can define:

$$\sigma(F) = \mathcal{C}^{-1}(\sigma_{\mathcal{C}}(F))$$

■

Theorem 1. *Decidability of second-order [context] unification can be NP-reduced to decidability of second-order [context] unification with just one binary function symbol, and constants*

Proof: By Lemmas 1 and 6, we know that, when variables do not touch, satisfiability of second-order and context unification problems is preserved by currying. Now, we will prove that we can NP-reduce solvability of second-order and context unification to solvability of the corresponding problems without touching variables.

For second-order the reduction is as follows. For every n-ary variable F, we conjecture one of the following possibilities:

– Project $F \mapsto \lambda x_1 \,.\, \ldots \lambda x_n \,.\, x_i$, for some $i \in \{1, \ldots, n\}$.
– Instantiate $F \mapsto \lambda x_1 \,.\, \ldots \lambda x_n \,.\, f(F_1(x_1, \ldots, x_n), \ldots, F_m(x_1, \ldots, x_n))$, for some constant $f \in \Sigma_m$ occurring in the original unification problem, and being $F_1, \ldots, F_m$ fresh free variables.

Obviously, this reduction can be performed in polynomial non-deterministic time. As far as the new problem is an instance of the original one, if the new problem is solvable, so it is the original one. If the original problem is solvable, and σ is a most general unifier, then, for every variable F, let $\sigma(F) = \lambda x_1 \ldots \lambda x_n \,.\, t$ be written in normal form. Taking t as a tree, descend from the root to the left-most leave, discarding free variables, until you get a constant f (this must be a constant occurring in the problem, by Property 1), a 0-ary variable, or a bound variable x_i. It is easy to prove that the instantiation $F \mapsto \lambda x_1 \ldots \lambda x_n \,.\, x_i$, if we find a bound variable x_i, $F \mapsto \lambda x_1 \ldots \lambda x_n \,.\, a$ for some fixed constant a, if we find a 0-ary variable, or $F \mapsto \lambda x_1 \ldots \lambda x_n \,.\, f(F_1(x_1, \ldots, x_n), \ldots, F_m(x_1, \ldots, x_n))$, if we find a constant f, results in a solvable problem. In fact, the solution of the new problem is σ composed with a substitution that projects as $G \mapsto \lambda x_1 \cdots \lambda x_n \,.\, x_1$ the free variables that we have discarded during the traversal, and as $X \mapsto a$ the 0-ary variables that we have found.

For context unification the reduction is as follows. For every n-ary variable F, we conjecture one of the following possibilities:

- Project $F \mapsto \lambda x \,.\, x$, if it is unary.
- Instantiate

$$F \mapsto \lambda x_1 \ldots \lambda x_n \,.\, f(F_1(x_{\tau(1)}, \ldots, x_{\tau(r_1)}), \ldots, F_m(x_{\tau(r_{m-1}+1)}, \ldots, x_{\tau(n)}))$$

  for some constant $f \in \Sigma_m$ occurring in the original unification problem, some permutation τ, and being $F_1, \ldots, F_m$ fresh free variables.

As for the second-order case, it can be proved that this nondeterministic reduction preserves satisfiability. However, in this case we have to assume that the original signature (the problem) contains, at least, a binary function symbol and a 0-ary constant.

■

6 Conclusions and Further Work

Currying terms is an standard technique in functional programming and has been used in practical applications of automated deduction. It is also used in higher-order unification via explicit substitutions or explicit unification. However, in these cases not only applications, but also lambda abstractions are made explicit, and unification is made modulo the explicit substitution rules. Here, we propose a partial currying transformation for second-order unification, where the "order" of the unification problem is not reduced, like in explicit unification, but the signature is simplified. The transformation is not trivial, and we prove that, to preserve solvability of the problems, we need to ensure that "variables do not touch". The reduction also works for context unification. This allows us to concentrate on a simpler signature containing constant symbols and just one binary function symbol: the explicit application symbol @.

Decidability of higher-order matching is still an open question. Proving that higher-order matching can be curried, i.e., that we can simplify the signature,

could contribute to prove its decidability or undecidability. The extension of our technique to third-order and higher orders is proposed as a further work.

The first difficulty we find trying to apply our transformation to third or higher order matching problems is that we must deal with instances of variables that are not connected. For instance, the following matching problem:

$$\begin{array}{l} f(\ F(\lambda x\,.g(x), a),\ F(\lambda x\,.g'(x), a')\ ) \stackrel{?}{=} \\ \stackrel{?}{=} f(\ f(g(h(a)), a),\quad f(g'(h(a')), a')\ ) \end{array}$$

is solved by the substitution:

$$F \mapsto \lambda x\, \lambda y\, .f(x, (h(y)), y)$$

where the instance of F is split into two pieces f and h. In such situations we have to guaranty that these pieces do not touch, to avoid that these "cuts" (in the sense of Plandowski [Pla99]) could cut a left chain of @'s.

Acknowledgments. We acknowledge Roberto Nieuwenhuis for suggesting us the use of currying in second-order unification problems. We also thank all the anonymous referees for their helpful comments.

References

[ACCL98] M. Abadi, L. Cardelli, P.-L. Curien, and J.J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1998.

[BM00] Nikolaj Bjorner and César Muñoz. Absoulte explicit unification. In *Proceedings of the 11th Int. Conf. on Rewriting Techniques and Applications (RTA'00)*, volume 1833 of *LNCS*, pages 31–46, Norwich, UK, 2000.

[Com98] Hubert Comon. Completion of rewrite systems with membership constraints. *Journal of Symbolic Computation*, 25(4):397–453, 1998.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. *Information and Computation*, 157:183–235, 2000.

[Far91] W. M. Farmer. Simple second-order languages for wich unification is undecidable. *Theoretical Computer Science*, 87:173–214, 1991.

[GNN01] Harald Ganzinger, Robert Nieuwenhuis, and Pilar Nivela. Context trees. In *Proceedings of the First Int. Conf. on Automated Reasoning (IJCAR 2001)*, volume 2083 of *LNCS*, pages 242–256, Siena, Italy, 2001.

[Gol81] W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

[Lev96] Jordi Levy. Linear second-order unification. In *Proceedings of the 7th Int. Conf. on Rewriting Techniques and Applications (RTA'96)*, volume 1103 of *LNCS*, pages 332–346, New Brunsbick, New Jersey, 1996.

[LV00] Jordi Levy and Margus Veanes. On the undecidability of second-order unification. *Information and Computation*, 159:125–150, 2000.

[LV01] Jordi Levy and Mateu Villaret. Context unification and traversal equations. In *Proceedings of the 12th Int. Conf. on Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *LNCS*, pages 167–184, Utrecht, The Netherlands, 2001.

[Mak77] G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. USSR Sbornik*, 32(2):129–198, 1977.

[Pla99] Wojciech Plandowski. Satisfiability of word equations with constants is in pspace. In *Proceedings of the 40th Annual Symposioum on Foundations of Computer Science, FOCS'99*, pages 495–500, New York, NY, USA, 1999.

[SS96] Manfred Schmidt-Schauß. An algorithm for distributive unification. In *Proceedings of the 7th Int. Conf. on Rewriting Techniques and Applications (RTA'96)*, volume 1103 of *LNCS*, pages 287–301, New Jersey, USA, 1996.

[SS98] Manfred Schmidt-Schauß. A decision algorithm for distributive unification. *Theoretical Computer Science*, 208:111–148, 1998.

[SSS98] Manfred Schmidt-Schauß and Klaus U. Schulz. On the exponent of periodicity of minimal solutions of context equations. In *Proceedings of the 9th Int. Conf. on Rewriting Techniques and Applications (RTA'98)*, volume 1379 of *LNCS*, pages 61–75, Tsukuba, Japan, 1998.

[SSS99] Manfred Schmidt-Schauß and Klaus U. Schulz. Solvability of context equations with two context variables is decidable. In *Proceedings of the 16th Int. Conf. on Automated Deduction (CADE-16)*, LNAI, pages 67–81, 1999.