Typed Unification: when failure may not be wrong

João Barbosa, Mário Florido, Vítor Santos Costa

Typed Unification: when failure may not be wrong

João Barbosa[1,2]*, Mário Florido[1,2] and Vítor Santos Costa[1]

[1] DCC, Faculdade de Ciências, Universidade do Porto
Rua do Campo Alegre s/n, 4169–007 Porto, Portugal
[2] Artificial Intelligence and Computer Science Laboratory – LIACC

Abstract

In a previous paper, we have investigated unification extended with regular types restricted to an Herbrand interpretation of term functors. Here we extend this investigation allowing arbitrary interpretations for functors. This new approach leads to a new typed unification algorithm which returns most general unifiers, failure or a wrong value denoting a type error during the unification process.

1 Introduction

Unification has been widely applied to different scenarios in computer science. Examples include database processing systems, natural language processing, computer vision, expert systems, computer algebra, programming language implementation, automated theorem proving, type inference and logic programming.

This paper is concerned with unification in typed first order theories, where types are described by regular grammars [8, 5, 7, 2, 3]. It is inspired by the fact that in logic programming many practical applications of variables, and the domains of functions, are naturally restricted to specific semantic domains which are subsets of the whole semantic universe.

With this motivation in mind we present a term unification algorithm which will return one of three different results: a *most general unifier*, *failure* or *wrong*. This last value *wrong* is inspired by a similar notion used by Robin Milner to denote run-time type errors [4] and, in our framework, it corresponds to the unification of terms that can never belong to the same semantic domains.

This paper generalizes a typed unification algorithm previously defined by the authors in [1] that was used in the dynamic typing of logic programs. In [1] functions symbols f of arity n, had co-domains which were always sets of terms of the form $f(t_1, \ldots, t_n)$ where the arguments t_i belong to the corresponding domain of f. This basically induced a partition of the Herbrand domain into sets of trees. We extend this notion by enabling the use of arbitrary semantic domains and co-domains for functors. A function now, may map integers to integers, integers to lists, floats to lists of integers, and, thus, semantic domains are no longer confined to be sets of trees.

Example 1: Let *cons* be the list constructor with type $cons :: \alpha \times list(\alpha) \rightarrow list(\alpha)$, where $list(\alpha) = nil + cons(\alpha, list(\alpha))$ (+ denotes disjoint type union). Suppose we have terms $t_1 = cons(1, X)$ and $t_2 = cons(Y, 1)$. These terms unify using first-order (untyped) unification, but they do not have a correct type, since the second argument of *cons* must be a list. This ill-typing is captured by our new typed unification algorithm that, in this case, outputs *wrong*.

*This work was financially supported by: Base Funding - UIDB/00027/2020 of the Artificial Intelligence and Computer Science Laboratory – LIACC - funded by national funds through the FCT/MCTES (PIDDAC).

The most obvious related work is many-sorted unification [6], though many-sorted unification, in general, assumes an infinite hierarchy of sorts and we do not assume a hierarchy of types. The most related unification problem is many-sorted unification with a forest-structured sort hierarchy [6], but even compared with this strong restricted unification problem, our work gives easier and nicer results, mostly due to the use of an expressive universe partition based on regular types but with no underlying hierarchy on the domains.

2 Language

The alphabet for the language of types includes an infinite set of type variables **TVAR**, a finite set of base types **TBASE**, an infinite set of type function symbols **TFUNC**, an infinite set of type symbols **TSYM**, parenthesis, and the comma. Then, a type term is either a type variable $\alpha \in$ **TVAR**, a base type $b \in$ **TBASE**, a type function symbol $f \in$ **TFUNC** applied to an n-tuple of type terms $f(\tau_1, \ldots, \tau_n)$ or a type symbol $\sigma \in$ **TSYM** applied to an n-tuple of type terms $\sigma(\tau_1, \ldots, \tau_n)$. If a type function symbol has arity 0, we call it a type constant. If the arity of a type function symbol is larger or equal to one, we call it a complex type term. In a complex type term we call the type function symbol that starts it the principal type functor. We use the same notation for complex type terms and complex terms, but from context it should be obvious to which one we are referring in each case.

Type symbols are defined by type definitions of the form: $\sigma(\alpha_1, \ldots, \alpha_n) = \tau_1 + \cdots + \tau_n$. The type variables on the left-hand side of the type definition are called parameters and correspond to all the variables that occur on the right-hand side. The sum $\tau_1 + \cdots + \tau_n$ is a *union type*, describing values that may have one of the types $\tau_1, \ldots, \tau_n$, called the summands. The '+' is an idempotent, commutative, and associative operation. Type definitions are called *deterministic* if no two type terms on the right-hand side have the same principal functor, and no type terms on the right-hand side start with a type symbol. Deterministic type definitions correspond to tuple-distributive types [8] and from now on we assume type definitions are deterministic.

We assume the universe of semantic values is separated into several disjoint domains $U = D_1 \cup \cdots \cup D_n$. Each domain is associated with a type. Note that since the domains are disjoint, if we have a certain type symbol σ defined by the type definition $\sigma(\alpha) = nil + cons(\alpha, \sigma(\alpha))$, we are assuming that no other type exists that uses the same type function symbols nor the same constants. We also assume that there is a one-to-one correspondence between function symbols and type function symbols, i.e., for each function symbol f there is a correspondent type function symbol f.

Let $t \in \tau$ mean that the semantic value denoted by t belongs to the domain associated with type τ. Let t be a ground term. If t is a constant c, then the type for $c :: \tau$, where τ is a base type b or a type symbol σ, if the type constant associated with c is a summand in the definition of σ. We say the principal type functor of c is τ.

If t is a complex term $f(t_1, \ldots, t_n)$, then we need to highlight the possibility of the term having a type error. In general, the type for f is $f :: \tau_1 \times \cdots \times \tau_n \to \tau$, where n is the arity of f, and τ is a base type, a complex type term, or a type symbol. If all t_i are such that $t_i \in \tau_i$, then there is no type error. But if at least one t_i is such that $t_i \notin \tau_i$ then we say that there is a type error. If τ is a type symbol σ then in the type definition of σ there is a summand of the form $f(\tau_1, \ldots, \tau_n)$. We also assume that the type for a variable is the top, a type representing the sum of all types, but we want to distinguish between different variables, and, more importantly, relate different variables, so we assume that each variable X_i is associated with a type variable α_i, that can be instantiated by any type, or remain as a variable, which can be interpreted as "any type".

3 Typed Unification Algorithm

We have three types of constraints. Equality between terms $t_1 = t_2$, equality between types $\tau_1 \doteq \tau_2$, and membership of terms in types $t \in \tau$.

Suppose we want to unify two terms t_1 and t_2. Let the type for the principal functor of t_1 and t_2 be τ_1 and τ_2, respectively. We build an initial tuple of sets of constraints of the form $(\{t_1 = t_2\}, \{t_1 \in \tau_1, t_2 \in \tau_2, \alpha \doteq \tau_1, \alpha \doteq \tau_2\})$, where α is a fresh type variable. Then the following rewriting system rewrites the tuple (C, T), where C are equality constraints that are generated in the algorithm, and T is a set of type constraints, both equality between types and membership of terms in types. The rules for the rewriting system are meant to be applied in order, i.e., if rule n and $n + k$ can both be applied, we apply n. They are as follows:

1. $(C, \{f(t_1, \ldots, t_n) \in \tau\} \cup Rest) \rightarrow (C, \{t_1 \in \tau\prime_1, \ldots, t_n \in \tau\prime_n, \tau\prime_1 \doteq \tau_1, \ldots, \tau\prime_n \doteq \tau_n\} \cup Rest)$, where the type for f is $f :: \tau_1 \times \cdots \times \tau_n \rightarrow \tau$, and $\tau\prime_i$ are the types for the principal functors of t_i, respectively
2. $(C, \{X_i \in \alpha_i\} \cup Rest) \rightarrow (C, Rest)$
3. $(C, \{c \in \tau\} \cup Rest) \rightarrow (C, Rest)$
4. $(C, \{f(\tau_1, \ldots, \tau_n) \doteq f(\tau\prime_1, \ldots, \tau\prime_n)\} \cup Rest) \rightarrow (C, \{\tau_1 \doteq \tau\prime_1, \ldots, \tau_n \doteq \tau\prime_n\} \cup Rest)$
5. $(C, \{\tau \doteq \tau\} \cup Rest) \rightarrow (C, Rest)$
6. $(C, \{f(\tau_1, \ldots, \tau_n) \doteq g(\tau\prime_1, \ldots, \tau\prime_m)\} \cup Rest) \rightarrow wrong$, if $f \neq g$ or $n \neq m$
7. $(C, \{\tau \doteq \alpha\} \cup Rest) \rightarrow (C, \{\alpha \doteq \tau\} \cup Rest)$, τ is not a type variable
8. $(C, \{\alpha \doteq \tau\} \cup Rest) \rightarrow (C, \{\alpha \doteq \tau\} \cup Rest[\alpha \mapsto \tau])$, if α does not occur in τ
9. $(C, \{\alpha \doteq \tau\} \cup Rest) \rightarrow wrong$, if α occurs in τ
10. $(\{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)\} \cup Rest, T) \rightarrow (\{t_1 = s_1, \ldots, t_n = s_n\} \cup Rest, T)$
11. $(\{t = t\} \cup Rest, T) \rightarrow (Rest, T)$
12. $(\{f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)\} \cup Rest, T) \rightarrow false$, if $f \neq g$ or $n \neq m$
13. $(\{t = X\} \cup Rest, T) \rightarrow (\{X = t\} \cup Rest, T)$, t is not a variable
14. $(\{X = t\} \cup Rest, T) \rightarrow (\{X = t\} \cup Rest[X \mapsto t], T)$, if X does not occur in t
15. $(\{X = t\} \cup Rest, T) \rightarrow false$, if X occurs in t.

We show a few examples of the application of this algorithm to pairs of terms.

Example 2: Suppose f has type $f :: int \times int \rightarrow int$, and 1 and 2 have type int. Then let $t_1 = f(1, f(X, 1))$ and $t_2 = f(Y, f(2, Y))$. The generated tuple of constraints is:

$(\{f(1, f(X, 1)) = f(Y, f(2, Y))\}, \{f(1, f(X, 1)) \in int, f(Y, f(2, Y)) \in int, int \doteq \alpha, int \doteq \alpha\}$.

Applying the rewriting rules, we get to:

$(\{f(1, f(X, 1)) = f(Y, f(2, Y))\}, \{f(1, f(X, 1)) \in int, f(Y, f(2, Y)) \in int, int \doteq \alpha\} \rightarrow^*$
$(\{f(1, f(X, 1)) = f(Y, f(2, Y))\}, \{X \in \alpha_1, Y \in \alpha_2, \alpha_2 \doteq int, \alpha_1 \doteq int, int \doteq int, int \doteq \alpha\} \rightarrow^*$
$(\{f(1, f(X, 1)) = f(Y, f(2, Y))\}, \{\alpha_2 \doteq int, \alpha_1 \doteq int, \alpha \doteq int\} \rightarrow^*$

$(\{Y = 1, X = 2\}, \{\alpha_2 \doteq int, \alpha_1 \doteq int, \alpha \doteq int\}$

The algorithm stops without failure and the most general unifier is $[Y \mapsto 1, X \mapsto 1]$.

Now an example where we get the result $false$.

Example 3: Suppose f has type $f :: int \times int \to int$, and 1 and 2 have type int. Then let $t_1 = f(1, f(1, X))$ and $t_2 = f(Y, f(2, Y))$. The generated tuple of constraints is:

$(\{f(1, f(1, X)) = f(Y, f(2, Y))\}, \{f(1, f(1, X)) \in int, f(Y, f(2, Y)) \in int, int \doteq \alpha, int \doteq \alpha\}$.
Applying the rewriting rules, we get to:
$(\{f(1, f(1, X)) = f(Y, f(2, Y))\}, \{f(1, f(1, X)) \in int, f(Y, f(2, Y)) \in int, int \doteq \alpha\} \to^*$
$(\{f(1, f(1, X)) = f(Y, f(2, Y))\}, \{X \in \alpha_1, Y \in \alpha_2, \alpha_2 \doteq int, \alpha_1 \doteq int, int \doteq int, int \doteq \alpha\} \to^*$
$(\{f(1, f(1, X)) = f(Y, f(2, Y))\}, \{\alpha_2 \doteq int, \alpha_1 \doteq int, \alpha \doteq int\} \to^*$
$(\{1 = 2\}, \{\alpha_2 \doteq int, \alpha_1 \doteq int, \alpha \doteq int\}$

The algorithm halts and outputs the result $false$.

And, finally, an example where we get the result $wrong$.

Example 4: Suppose f has type $f :: int \times int \to int$, g has type $g :: int \to float$, and 1 and 2 have type int. Then let $t_1 = f(1, f(1, X))$ and $t_2 = f(Y, g(2))$. The generated tuple of constraints is:

$(\{f(1, f(1, X)) = f(Y, g(2))\}, \{f(1, f(1, X)) \in int, f(Y, g(2)) \in int, int \doteq \alpha, int \doteq \alpha\}$.
Applying the rewriting rules, we get to:
$(\{f(1, f(1, X)) = f(Y, g(2))\}, \{f(1, f(1, X)) \in int, f(Y, g(2)) \in int, int \doteq \alpha\} \to^*$
$(\{f(1, f(1, X)) = f(Y, g(2))\}, \{X \in \alpha_1, Y \in \alpha_2, \alpha_2 \doteq int, \alpha_1 \doteq int, int \doteq int, float \doteq int, int \doteq \alpha\} \to^*$
$(\{f(1, f(1, X)) = f(Y, g(2))\}, \{\alpha_2 \doteq int, \alpha_1 \doteq int, float \doteq int, \alpha \doteq int\}$
The algorithm halts and outputs the result $wrong$.

3.1 Termination

Here we show that, for any input, the unification algorithm always terminates.

Theorem 1 - Termination: For any t_1 and t_2, let (C, T) be the set of generated constraints from the terms. The rewrite system always terminates, returning a most general unifier, false, or wrong.

Proof. We divide the proof into three parts. Firstly, we prove that the number of membership constraints reduces to zero, from rules 1 through 3. Afterwards, we prove the termination of the rewriting algorithm for the type equality constraints, from rules 4 through 9. Then we prove the termination of the rewriting algorithm for equality constraints, from rules 10 through 15. Since the algorithm uses the rules in order and no membership constraints are generated after we start applying steps 4 through 9, such as no type equality constraints are generated after we start applying rules 10 through 15, we prove the termination of the algorithm.

The proof for the first part follows a usual termination proof approach, where we show that a carefully chosen metric decreases at every step.

The metric is the number of symbols on the left-hand side of constraints. All rules 1 through 3 decrease this metric.

The second and third parts are each the Martelli-Montanari algorithm for its corresponding kind of constraints, type equality and equality, respectively. Therefore they also terminate, since the Martelli-Montanari algorithm terminates. □

Another important property is that the algorithm is sound. We conjecture this property, because we have not completed the prove yet (it is ongoing work).

Theorem 2 - Soundness:

1) Suppose our algorithm returns a unifier μ. Then μ is a most general unifier and any substitution $\theta = \mu \circ \lambda$ that grounds terms t_1 and t_2 is such that $\theta(t_1) = \theta(t_2)$ and $\theta(t_i)$ is well-typed.

2) Suppose our algorithm outputs *wrong*. Then, there is no substitution θ that grounds terms t_1 and t_2 such that $\theta(t_1)$ and $\theta(t_2)$ have the same type.

3) Suppose our algorithm outputs $false$. Then, there is no substitution θ such that $\theta(t_1) = \theta(t_2)$, but there is a substitution $\theta\prime$, such that $\theta\prime(t_1)$ and $\theta\prime(t_2)$ have the same type.

References

[1] João Barbosa, Mário Florido, and Vítor Santos Costa. Typed SLD-Resolution: Dynamic typing for logic programming. In Alicia Villanueva, editor, *Logic-Based Program Synthesis and Transformation*, pages 123–141, Cham, 2022. Springer International Publishing.

[2] Thom W. Frühwirth, Ehud Y. Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proc. of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Netherlands,1991*, pages 300–309, 1991.

[3] John P Gallagher and Kim S Henriksen. Abstract Domains Based on Regular Types. In *International Conference on Logic Programming*, pages 27–42. Springer, 2004.

[4] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[5] Prateek Mishra. Towards a theory of types in Prolog. In *Proceedings of the 1984 International Symposium on Logic Programming, Atlantic City, New Jersey, USA, February 6-9, 1984*, pages 289–298. IEEE-CS, 1984.

[6] Christoph Walther. Many-sorted unification. *J. ACM*, 35(1):1–17, jan 1988.

[7] Eyal Yardeni and Ehud Shapiro. A type system for logic programs. *The Journal of Logic Programming*, 10(2):125–153, 1991.

[8] Justin Zobel. Derivation of polymorphic types for Prolog programs. In *Logic Programming, Proceedings of the Fourth International Conference, Melbourne, 1987*, 1987.