

Regular Typed Unification

João Barbosa

*Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto
Rua do Campo Alegre s/n, 4169-007 Porto, Portugal
LIACC - Laboratório de Inteligência Artificial e Ciência de Computadores*

Mário Florido

*Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto
Rua do Campo Alegre s/n, 4169-007 Porto, Portugal
LIACC - Laboratório de Inteligência Artificial e Ciência de Computadores*

Vítor Santos Costa

*Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto
Rua do Campo Alegre s/n, 4169-007 Porto, Portugal
INESCTEC - Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência*

Abstract

Here we define a new unification algorithm for terms interpreted in semantic domains denoted by a subclass of regular types here called deterministic regular types. This reflects our intention not to handle the semantic universe as a homogeneous collection of values, but instead, to partition it in a way that is similar to data types in programming languages. We first define the new unification algorithm which is based on constraint generation and constraint solving, and then prove its main properties: termination, soundness, and completeness with respect to the semantics. Finally, we discuss how to apply this algorithm to a dynamically typed version of Prolog.

KEYWORDS: Regular Types, Unification, Typed Unification

1 Introduction

In mathematical logic, a *term* denotes a mathematical object. In logic programming this notion is generalized to terms as a syntactic representations of structured data. A theory of equality on the set of all terms formally defines which terms are considered equal. In the most typical case of first order theories only syntactically identical terms are considered equal. In this case functions are uninterpreted functions or, computationally, functions build data terms, rather than operating on them.

When the domain of discourse contains elements of different kinds, it is useful to split the set of all terms (Universe) accordingly. To this end, a *type* (sometimes also called *sort*) is assigned to variables and constant symbols, and a declaration of the domain type and range type to each function symbol. A typed term $f(t_1, \dots, t_n)$ may then be composed from the subterms t_1, \dots, t_n only if the i -th subterm's type matches the declared i -th domain type of f . Such a term is called *well-typed* and terms which are not well-typed are called *ill-typed*.

Previous approaches for types in logic programming use *regular types* as the type language for logic programming Zobel (1987); Mishra (1984); Yardeni and Shapiro (1991); Frühwirth et al. (1991); Dart and Zobel (1992); Codish and Lagoon (2000); Schrijvers et al. (2008b;a); Gallagher and Henriksen (2004); Barbosa et al. (2022a); Hermenegildo et al. (2023). A subset of regular types, which restricts individual type definitions to be deterministic, allows for decidable emptiness checking, subset checking, and a intersection and unification operations as defined by Dart and Zobel Dart and Zobel (1992). However, when trying to find a semantic partition corresponding to regular types (even with the same restriction), type checking is not trivial.

Data type definitions in programming languages impose constraints to the type language to allow decidable type checking. Usually data types are possibly recursive definitions where type constructors are unique. Here we propose a stricter subset of regular types that we shall call *deterministic regular types*, which corresponds exactly to the restriction mentioned above. In a deterministic regular type each type constructor (here called *type function symbol*) is unique.

We will focus our attention on equality in a theory where the semantic universe is partitioned in a way consistent with deterministic regular types, similarly to data types in programming languages. More specifically, this paper is concerned with unification in typed first order theories, where types are described by deterministic regular types.

With this motivation in mind we present a term unification algorithm which may now return 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 in functional programs Milner (1978) and, in our framework, it corresponds to the unification of terms that can never belong to the same semantic domains.

A function now, may map integers to integers, integers to lists, floats to lists of integers, and, thus, the Herbrand universe is now divided in many different domains.

Example 1: Let *cons* be the list constructor with type $cons :: \forall \alpha. \alpha \times list(\alpha) \rightarrow list(\alpha)$, where $list(\alpha) = [] + [\alpha \mid list(\alpha)]$ (+ denotes disjoint type union). Suppose we have terms $t_1 = cons(1, X)$ and $t_2 = cons(Y, 2)$. 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*.

Related Work This paper generalizes a typed unification algorithm previously defined by the authors in Barbosa et al. (2022b) that was used in the dynamic typing of logic programs. In Barbosa et al. (2022b) functions symbols f of arity n had co-domains which were always sets of terms of the form $f(t_1, \dots, t_n)$, 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. Here we extend this notion enabling the use of semantic domains and co-domains described by deterministic regular types.

The most obvious related work is many-sorted unification Walther (1988), though many-sorted unification assumes an infinite hierarchy of sorts and we do not assume a hierarchy of types. In particular there is a relation with many-sorted unification with a forest-structured sort hierarchy Walther (1988), 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 deterministic regular types but with no underlying hierarchy on the domains.

Here we study unification of terms interpreted in domains described by deterministic regular types, and we allow a form of parametric polymorphism in the description of term variables. Parametric polymorphic descriptions of sorted domains goes back to Smolka generalized order-sorted logic Smolka (1988). In his system, subsort declarations are propagated to complex type expressions, thus the main focus is on subtyping which is not the scope of our work.

Contributions Our main contributions are: an extension to the semantics defined in Barbosa et al. (2022b), where equality takes into account the domains of the two terms in the left and the right hand side of an equation, being *wrong* when terms belong to disjoint domains; a *type system* for terms and the equality predicate which we prove to be sound with respect to the semantic typing relation; and a new unification algorithm, which given an equation between two terms returns their most general unifier and their *principal type*, if there is a solution, *false* if there is not a solution but terms can belong to the same semantic domain and *wrong* otherwise.

This three stage framework (first a notion of semantic typing, then a type system for terms and equations which is sound with respect to semantic typing and, finally, a unification algorithm which is sound and complete with respect to the type systems) enables to smoothly prove soundness and completeness of our unification algorithm, and it is inspired by the type theory in Milner Milner (1978) where he also defined a notion of semantic typing, then a type system which was sound with respect to semantic typing and a type inference algorithm which was sound and complete with respect to the type system.

2 Term Syntax and Semantics

We will, firstly, define the language of terms. For a more in detail presentation, we refer the reader to Apt (1996); Lloyd (1984).

Given an infinite set of variables **VAR** and an infinite set of function symbols **FUNC**, a term is:

1. a variable (X, Y, X_i, \dots);
2. a function symbol of arity 0 ($k, a, b, 1, \dots$), which we call a constant;
3. a function symbol of arity $n \geq 1$ (f, g, h, \dots) applied to an n -tuple of terms.

We call terms that contain no variables *ground terms*, and terms that start with a function symbol with arity $n \geq 1$ *complex terms*.

Following the standard Herbrand interpretation of logic programs Apt (1996); Lloyd (1984), we assume that every term represents a tree and that all these trees are part of the universe of interpretation of the logic program.

Any division of the universe leads to a typed unification algorithm, but here, we assume a particular partition of the universe into several domains. Note that this interpretation just groups sets of trees in the universe into domains, and includes some other domains that are not consisting of trees. Thus, here we divide the universe **U** into domains as follows:

$$\mathbf{U} = \mathbf{Int} \cup \mathbf{Flt} \cup \mathbf{Str} \cup \mathbf{Atm} \cup \mathbf{List}_1 \cup \dots \cup \mathbf{List}_n \cup \mathbf{A}_1 \cup \dots \cup \mathbf{A}_m \cup \mathbf{Bool} \cup \mathbf{F} \cup \mathbf{Wrong},$$

where **Int** is the set of trees that represent integers (examples include 1 and -10, but also trees such as $1 + 4$ and $2 * 5 - 1$), **Flt** is the set of trees that represent floating-point numbers (similarly to **Int**, we consider every tree that represents a floating-point number, including trees whose root is an arithmetic operator), **Str** is the set of trees representing strings, **Atm** is the set of trees consisting of a single node, the root, that are not included in any other domain, **List_i** are sets of trees that represent lists, where each domain contains the trees that represent lists of elements of some other domain (i.e., we have a domain for lists of integers, lists of strings, lists of lists of integer, ...), **A_i** are the domains of trees whose root is a function symbol and the nodes of each tree are in the same domain as the corresponding nodes of every other tree (examples: $f(\mathbf{Int})$, $g(\mathbf{Int}, \mathbf{Float})$, $h(g(\mathbf{Atom}), h(\mathbf{Int}))$, ...), **Bool** is the set with *true* and *false*, **F** is the set of semantic functions, and **Wrong** is the set with a single value, *wrong*. We call base domains to the domains **Int**, **Flt**, **Str**, and **Atm**.

One important note here is the value $[]$, corresponding to the empty list. We assume that this value belongs to every list domain, and that it is the only value that belongs to more than one domain in this partition.

Then, the semantics of a term is a tree in some domain, or *wrong*. The semantics depends on an interpretation **I** for the function symbols in the language, and a state **Σ** which associates variables to semantic values. We assume that the value returned by **I** is, for constants, a tree with just a root, and for function symbols of arity $n \geq 1$ a function in **F**, that outputs a tree and has as input some domains. We assume that, for all function symbols **f** except the list constructor **cons**, the corresponding function in **I** is a function f that has signature $f : \forall \alpha_1, \dots, \alpha_n. \alpha_1 \times \dots \times \alpha_n \rightarrow f(\alpha_1, \dots, \alpha_n)$, such that if any of the arguments the function is applied to is *wrong* then it outputs *wrong*, otherwise it outputs the tree with root f and children the trees it got has input. For the list constructor **cons** the function associated in **I** is *cons* with signature $cons : \forall \alpha. \alpha \times list(\alpha) \rightarrow list(\alpha)$ defined as:

$$cons(v_1, v_2) = \begin{cases} cons(v_1, v_2) & \text{if } v_1 \in D \wedge v_2 \in List(D) \\ wrong & \text{otherwise} \end{cases}$$

The predefined interpretation I is the one where every constant has the expected value, for instance the term 1 has as value the integer 1, and the term a has as value the atom **a**. One additional useful definition is the function *dom* that returns what is the domain of a value.

We define the semantics of a term, represented by $\llbracket \cdot \rrbracket_{I, \Sigma}$, in the following way:

- $\llbracket X \rrbracket_{I, \Sigma} = \Sigma(X)$
- $\llbracket k \rrbracket_{I, \Sigma} = I(k)$
- $\llbracket f(t_1, \dots, t_n) \rrbracket_{I, \Sigma} = I(f)(\llbracket t_1 \rrbracket_{I, \Sigma}, \dots, \llbracket t_n \rrbracket_{I, \Sigma})$

Note that, if a complex term contains the list constructor, the semantics of that term can be *wrong*. This is where the division into domains comes into play, since if we were considering an undivided Herbrand universe, then trivially all values are in the same domain so the application of a function could never generate an error. This is not

realistic in any typed programming language, but also true for untyped languages: note that Prolog, the main logic programming language, already includes some division of the universe into values, implicit by the use of type predicates such as *integer*, *float* or *atom*.

We assume one predicate, equality, hereby represented by $=$. The semantics of equality is re-adjusted to take into account the value *wrong*. Equality is defined for terms in the same domain. So let function *eq* with signature $eq : \forall \alpha. \alpha \times \alpha \rightarrow Bool$ be defined as follows:

$$eq(v_1, v_2) = \begin{cases} true & \text{if } v_1 = v_2 \wedge dom(v_1) \cap dom(v_2) \neq \emptyset \wedge dom(v_1) \neq Wrong \\ false & \text{if } v_1 \neq v_2 \wedge dom(v_1) \cap dom(v_2) \neq \emptyset \wedge dom(v_1) \neq Wrong \\ wrong & \text{otherwise} \end{cases}$$

The semantics for the equality predicate is then as follows:

$$\llbracket t_1 = t_2 \rrbracket_{I, \Sigma} = eq(\llbracket t_1 \rrbracket_{I, \Sigma}, \llbracket t_2 \rrbracket_{I, \Sigma})$$

Note that there are two ways of returning *wrong* in an equality: either one of the terms contains an error in itself (i.e., has the semantic value *wrong*), therefore it belongs to the domain **Wrong**, or the semantics for the terms belongs to domains with an empty intersection. The second case needs some explanation. We could define it as $dom(v_1) \neq dom(v_2)$, but for values that belong to several domains, such as $[]$, the *dom* function returns the union of all the domains, so equality is well defined when one finds a common domain for both terms.

3 Type Language

Here we define a syntactic description of semantic domains. This syntactic description corresponds to the well know concept of a *type*. 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. There is a one-to-one correspondence between **TFunc** and **Func**, which we assume is predefined. Then, we have the following grammar for types:

$$\begin{aligned} all_type &::= cons_type \mid func_type \\ cons_type &::= type \mid type_term \\ func_type &::= type_1 \times \dots \times type_n \rightarrow type \\ type &::= tvar \mid tbase \mid tsymbol(type_1, \dots, type_n) \\ type_term &::= tconstant \mid tfunction(cons_type_1, \dots, cons_type_n) \\ type_def &::= tsymbol(tvar_1, \dots, tvar_n) \longrightarrow type_term_1 + \dots + type_term_m \end{aligned}$$

Where $tvar \in \mathbf{TVar}$, $tbase \in \mathbf{TBase}$, $tconstant, tfunction \in \mathbf{TFunc}$, and $tsymbol \in \mathbf{TSym}$. We call a type term that starts with a *tfunction* a complex type term. We call *ground* to any type that does not contain a type variable.

Each type symbol is defined by a *type definition*. A *well-formed* type definition has all *tvar* that occur as parameters on the left-hand side of the definition, occurring somewhere on the right-hand side. The sum $\tau_1 + \dots + \tau_n$ is a *union type*, describing values that have one of the type terms τ_1, \dots, τ_n , called the *summands*. The ‘+’ is an idempotent, commutative, and associative operation.

A set of type definitions D is called *deterministic* if any type function symbol occurs at most once in D . Note that in a deterministic set of type definitions no two type terms on the right-hand side of a definition have the same principal functor. Deterministic type definitions include tuple-distributive types Zobel (1987) and correspond to the widely used algebraic data types in programming languages. From now on we assume that type definitions are deterministic and well-formed.

A *type scheme* σ is either a type, a functional type or an expression of the form $\forall \alpha_1, \dots, \alpha_n. \tau$, where τ is an *all_type* and $\alpha_1, \dots, \alpha_n$ are type variables which will be called the *generic variables* of σ . Note that types form a subclass of type schemes. To simplify presentation, we will often abbreviate type schemes to $\forall \vec{\alpha}. \tau$, where $\vec{\alpha}$ denotes a sequence of several type variables α_i . Type schemes represent parametric polymorphic types Damas and Milner (1982).

4 Semantics

Each type is associated with a domain. A base type *tbase* is associated with a base domain, and each instance of a type of the form $tsymbol(type_1, \dots, type_n)$ is associated with a domain. In our language we include a single type symbol *list* that is associated with the domains for lists. We assume that the definition for the type symbol *list* is: $list(\alpha) \longrightarrow [] + cons(\alpha, list(\alpha))$. We could include further type symbols that were defined by inductive definitions, such as lists, and the rest of this paper could be easily extended to include different inductively defined types, but we keep *list* as the only one in this paper for the sake of simplicity. We assume that every type term on the right-hand side of a type definition is a type constant or a complex type term, and the definitions are deterministic (each type constant and type function symbol occurs only once).

A valuation ψ gives a ground type for each type variable. Formally, the semantics of a type is a domain. Given a valuation ψ , we define the semantics of a *type* as follows:

$$\begin{aligned}
\mathbf{T}[\alpha]_\psi &= \mathbf{T}[\psi(\alpha)]_\psi \\
\mathbf{T}[int]_\psi &= \mathbf{Int} \\
\mathbf{T}[float]_\psi &= \mathbf{Flt} \\
\mathbf{T}[string]_\psi &= \mathbf{Str} \\
\mathbf{T}[atom]_\psi &= \mathbf{Atm} \\
\mathbf{T}[list(\alpha)]_\psi &= \mathbf{T}[list(\psi(\alpha))]_\psi \\
\mathbf{T}[list(int)]_\psi &= List(Int), \text{ same for any other ground instance of } list(\alpha) \\
\text{The semantics of a } type_term &\text{ is as follows, also given a valuation } \psi: \\
\mathbf{T}[k]_\psi &= \{k\} \\
\mathbf{T}[f(\tau_1, \dots, \tau_n)]_\psi &= \{f(v_1, \dots, v_n) \mid v_i \in \mathbf{T}[\tau_i]_\psi\}
\end{aligned}$$

The semantics of a *union_type*, given a valuation ψ , is:

$$\mathbf{T}[\tau_1 + \dots + \tau_n]_\psi = \mathbf{T}[\tau_1]_\psi \cup \dots \cup \mathbf{T}[\tau_n]_\psi$$

The semantics of a *func_type*, given a valuation ψ , is:

$$\mathbf{T}[\tau_1 \times \dots \times \tau_n \rightarrow \tau]_\psi = \{f \mid f \in \mathbf{F} \wedge \forall v_1 \in \mathbf{T}[\tau_1]_\psi, \dots, v_n \in \mathbf{T}[\tau_n]_\psi. f(v_1, \dots, v_n) \in \mathbf{T}[\tau]_\psi\}$$

And, finally, the semantics of a *type scheme* is:

$\mathbf{T}[\llbracket \forall \vec{\alpha}. \tau \rrbracket]_\psi = \bigcap_{\vec{\sigma}} \mathbf{T}[\llbracket \tau[\vec{\alpha} \mapsto \vec{\sigma}] \rrbracket]_\psi$, where $\vec{\sigma}$ is a sequence of types of the same size as $\vec{\alpha}$.

Note that the semantics of a (ground) *type-term* may be a domain, as in the case of $f(int, float)$, or the subset of a domain, as in the case of $cons(int, [])$, or even a subset of several domains, as in the case of $[]$. This includes a subset of the domain **Wrong**, as in the case of $cons(int, int)$. All instances of complex type terms whose *tfunction* is not the list constructor are associated with a domain for trees.

Also note that we assume a function, given by I , for the interpretation of function symbols, thus functions have type signatures: the type of a function symbol f of arity n is interpreted as function which builds a tree of root f , with the type scheme $\forall \alpha_1, \dots, \alpha_n. \alpha_1 \times \dots \times \alpha_n \rightarrow f(\alpha_1, \dots, \alpha_n)$. The semantics for this type scheme is the intersection of the semantics for all instances of the functional type, which is a subset of **F** consisting of all functions that have such type. So it consists of all the functions that can have any tuple of n elements as input and output a tree whose root is f and the children nodes are the input elements.

4.1 Semantic Typing

We now define what it means for a term to semantically have a type, denoted by $t : \tau$. If the term and the type are both ground, given an interpretation I , we just check whether the semantics of the term belongs to the domain corresponding to the semantics of the type. So, for ground terms and types:

$$t : \tau \implies \forall \Sigma. \forall \psi. \llbracket t \rrbracket_{I, \Sigma} \in \mathbf{T}[\llbracket \tau \rrbracket]_\psi$$

However, both terms and types can be non-ground in general and, without extra information, we cannot know what is the correct type for a variable. To deal with variables we introduce the concept of a context Γ , where we have typings of the form $X : \tau$ for variables. Now, given a context associating variables with types, we are able to define the *semantic typing* relation, denoted by \Vdash , defined as:

$$\Gamma \Vdash_I t : \tau \implies \forall \Sigma. \forall \psi. (\forall (X : \tau') \in \Gamma. \llbracket X \rrbracket_{I, \Sigma} \in \mathbf{T}[\llbracket \tau' \rrbracket]_\psi \implies \llbracket t \rrbracket_{I, \Sigma} \in \mathbf{T}[\llbracket \tau \rrbracket]_\psi)$$

Contexts allow us to type non-ground terms, assuming that the types for each variable are the ones given in the context. So the same term may have different types depending on the context, if it is non-ground (since ground terms always have the same type, due to the nonexistence of variables in the term that depend on the context). We call the *generic context* to the context that contains $X_i : \alpha_i$, for all variables, i.e., all variables have the most general type they can have, a type variable, and each type variable is associated with a particular term variable.

Example 2: Let $\Gamma = \{X : \alpha, Y : list(\alpha)\}$ and $\Delta = \{cons : \forall \alpha. \alpha \times list(\alpha) \rightarrow list(\alpha)\}$.

$$\Gamma, \Delta \models_I cons(X, Y) : list(\alpha)$$

Suppose we have a state Σ and a valuation ψ such that $\llbracket X \rrbracket_{I, \Sigma} \in \mathbf{T}[\llbracket \alpha \rrbracket]_\psi$ and $\llbracket Y \rrbracket_{I, \Sigma} \in \mathbf{T}[\llbracket list(\alpha) \rrbracket]_\psi$, then $\llbracket cons(X, Y) \rrbracket_{I, \Sigma} \in \mathbf{T}[\llbracket list(\alpha) \rrbracket]_\psi$. Since $\llbracket X \rrbracket_{I, \Sigma} = \Sigma(X) \in \mathbf{T}[\llbracket \psi(\alpha) \rrbracket]_\psi$ and

$\llbracket Y \rrbracket_{I, \Sigma} = \Sigma(Y) \in \mathbf{T}[\llbracket \psi(\text{list}(\alpha)) \rrbracket_\psi]$, by the semantics of **cons**, we have $\text{cons}(\Sigma(X), \Sigma(Y))$, which is not *wrong* from the domains of the respective values, and because the output is in the correct domain.

However, note that for $\Gamma = \{X : \alpha, Y : \beta\}$, the same would not be true, since for $\Sigma = [X \mapsto 1, Y \mapsto 2]$ and $\psi = [\alpha \mapsto \text{int}, \beta \mapsto \text{int}]$, the left-hand side of the implication is true, but $\text{cons}(1, 2)(= \text{wrong}) \notin \mathbf{T}[\llbracket \text{list}(\text{Int}) \rrbracket_\psi]$.

5 Syntactic Typing

This section introduces *syntactic typing* by the definition of a type system for terms and the equality predicate. A context Γ and a set of type assumptions for constants and function symbols Δ are needed to derive a type assignment and one writes $\Gamma, \Delta \vdash t : \tau$ (pronounce this as Γ and Δ yield t in τ). Assumptions in Δ are of the form $k : \forall \vec{\alpha}. \tau$, for constants, and $f : \forall \vec{\alpha}. \tau_1 \times \dots \times \tau_n \rightarrow \tau$, for function symbols, where the generic variables $\vec{\alpha}$ of these type schemes are exactly the type variables that occur in τ and $\tau_1 \times \dots \times \tau_n \rightarrow \tau$, respectively.

A statement $t : \tau$ is *derivable* from contexts Γ and Δ , notation $\Gamma, \Delta \vdash t : \tau$, if $\Gamma, \Delta \vdash t : \tau$ can be produced by the following rules.

$$\begin{array}{c}
 \text{VAR} \frac{(X : \tau) \in \Gamma}{\Gamma, \Delta \vdash X : \tau} \qquad \text{CST} \frac{(k : \forall \vec{\alpha}. \tau) \in \Delta}{\Gamma, \Delta \vdash k : \tau[\vec{\alpha} \mapsto \vec{\sigma}]} \\
 \\
 \text{CPL} \frac{(f : \forall \vec{\alpha}. \tau_1 \times \dots \times \tau_n \rightarrow \tau) \in \Delta \quad \Gamma, \Delta \vdash t_1 : \tau_1[\vec{\alpha} \mapsto \vec{\sigma}] \quad \dots \quad \Gamma, \Delta \vdash t_n : \tau_n[\vec{\alpha} \mapsto \vec{\sigma}]}{\Gamma, \Delta \vdash f(t_1, \dots, t_n) : \tau[\vec{\alpha} \mapsto \vec{\sigma}]} \qquad \text{EQU} \frac{\Gamma, \Delta \vdash t_1 : \tau \quad \Gamma, \Delta \vdash t_2 : \tau}{\Gamma, \Delta \vdash t_1 = t_2 : \text{bool}}
 \end{array}$$

Fig. 1. Type System

The type system definition pays particular attention to the mode of the typing judgment. In particular, the type contexts and term are interpreted as inputs to the typing judgment, while the term's type is viewed as an output.

We must guarantee that Δ is in agreement with I . For this, we have the following relation: $\text{Sig}_I \Vdash \Delta$, which is defined as $\forall (k : \tau) \in \Delta. \text{dom}(I(k)) = \tau \wedge \forall (f : \tau_1 \times \dots \times \tau_n \rightarrow \tau). I(f) : \tau_1 \times \dots \times \tau_n \rightarrow \tau$. We purposefully overload the symbol \Vdash since this corresponds to a semantic validation of a syntactic typing.

Example 3: Let $\Gamma = \{X : \text{int}, Y : \text{list}(\text{int})\}$, $\Delta = \{1 : \text{int}, \text{nil} : \forall \gamma. \text{list}(\gamma), \text{cons} : \forall \beta. \beta \times \text{list}(\beta) \rightarrow \text{list}(\beta)\}$, and $\Lambda = (\text{cons} : \forall \beta. \beta \times \text{list}(\beta) \rightarrow \text{list}(\beta)) \in \Delta$. Then the following type derivation holds using the type rules:

$$\frac{
 \frac{(X : \text{int}) \in \Gamma}{\Gamma, \Delta \vdash X : \text{int}} \quad
 \frac{([\] : \forall \gamma. \text{list}(\gamma)) \in \Delta}{\Gamma, \Delta \vdash [\] : \text{list}(\text{int})^{(2)}} \quad \Lambda
 }{
 \Gamma, \Delta \vdash \text{cons}(X, [\]) : \text{list}(\text{int})^{(1)}
 } \quad
 \frac{
 \frac{(1 : \text{int}) \in \Delta}{\Gamma, \Delta \vdash 1 : \text{int}} \quad
 \frac{(Y : \text{list}(\text{int})) \in \Gamma}{\Gamma, \Delta \vdash Y : \text{list}(\text{int})} \quad \Lambda
 }{
 \Gamma, \Delta \vdash \text{cons}(1, Y) : \text{list}(\text{int})^{(2)}
 }$$

$$\Gamma, \Delta \vdash \text{cons}(X, [\]) = \text{cons}(1, Y) : \text{bool}$$

Note that in ⁽¹⁾ we used $\text{list}(\beta)[\beta \mapsto \text{int}]$ and in ⁽²⁾ we used $\text{list}(\gamma)[\gamma \mapsto \text{int}]$. Also note that if $X : \alpha$ instead of $X : \text{int}$ was in Γ , we could not have a derivation.

We now prove that the rules for syntactic typing are sound, that is, if the set Δ is in agreement with I , then any type derivation is semantically correct.

Theorem 1 - Soundness of Syntactic Typing: If $\Gamma, \Delta \vdash t : \tau$ and $\text{Sig}_I \models \Delta$, then $\Gamma \Vdash_I t : \tau$.

Proof

We will prove this by induction on the derivation.

- If the term t is a variable X , then the derivation consists of a single application of axiom *VAR*. Clearly, it is also true that $\Gamma \Vdash_I X : \tau$, where $(X : \tau) \in \Gamma$, since any Σ that gives values to X and ψ that gives values to τ , such that $\llbracket X \rrbracket_{I, \Sigma} \in \mathbf{T}[\llbracket \tau \rrbracket_\psi]$ will do so in the context and in the term itself simultaneously, so $\Gamma \Vdash_I X : \tau$.
- If the term t is a constant k , then the derivation consists of a single application of axiom *CST*. Since $\text{Sig}_I \models \Delta$, $\text{dom}(I(k)) = \forall \vec{\alpha}. \tau$, where $(k : \forall \vec{\alpha}. \tau) \in \Delta$, then $k \in \mathbf{T}[\llbracket \forall \vec{\alpha}. \tau \rrbracket_\psi]$, for any ψ . But since $\mathbf{T}[\llbracket \forall \vec{\alpha}. \tau \rrbracket_\psi] = \bigcap_{\forall \vec{\sigma}} \mathbf{T}[\llbracket \tau[\vec{\alpha} \mapsto \vec{\sigma}] \rrbracket_\psi]$, then $k \in \mathbf{T}[\llbracket \tau[\vec{\alpha} \mapsto \vec{\sigma}] \rrbracket_\psi]$. So for any Σ , the right-hand side of the implication is always true, so $\Gamma \Vdash_I k : \tau[\vec{\alpha} \mapsto \vec{\sigma}]$.
- If the term t is a complex term $f(t_1, \dots, t_n)$, then we can assume, by induction hypothesis, that $\Gamma, \Delta \vdash t_i : \tau_i[\vec{\alpha} \mapsto \vec{\sigma}]$, for all $i = 1, \dots, n$. Since $\text{Sig}_I \models \Delta$, $I(f) : \forall \vec{\alpha}. \tau_1 \times \dots \times \tau_n \rightarrow \tau$, then $f \in \mathbf{T}[\llbracket \forall \vec{\alpha}. \tau_1 \times \dots \times \tau_n \rightarrow \tau \rrbracket_\psi]$, for all ψ , so $f \in \mathbf{T}[\llbracket (\tau_1 \times \dots \times \tau_n \rightarrow \tau)[\vec{\alpha} \mapsto \vec{\sigma}] \rrbracket_\psi]$. Therefore, we know that, if $v_i \in \mathbf{T}[\llbracket \tau_i[\vec{\alpha} \mapsto \vec{\sigma}] \rrbracket_\psi]$ then $f(v_1, \dots, v_n) \in \mathbf{T}[\llbracket \tau[\vec{\alpha} \mapsto \vec{\sigma}] \rrbracket_\psi]$. For any Σ and ψ such that $\forall (X : \tau_i) \in \Gamma. \llbracket X \rrbracket_{I, \Sigma} \in \mathbf{T}[\llbracket \tau_i \rrbracket_\psi]$, we know by the induction hypothesis $\llbracket t_i \rrbracket_{I, \Sigma} \in \mathbf{T}[\llbracket \tau_i[\vec{\alpha} \mapsto \vec{\sigma}] \rrbracket_\psi]$. Therefore, for the same Σ and ψ , we know that $\llbracket f(t_1, \dots, t_n) \rrbracket_{I, \Sigma} \in \mathbf{T}[\llbracket \tau[\vec{\alpha} \mapsto \vec{\sigma}] \rrbracket_\psi]$, so $\Gamma \Vdash_I f(t_1, \dots, t_n) : \tau[\vec{\alpha} \mapsto \vec{\sigma}]$.
- If we have an equality of two terms $t_1 = t_2$, we can assume, by induction hypothesis, that $\Gamma \Vdash_I t_1 : \tau$ and $\Gamma \Vdash_I t_2 : \tau$. Therefore we know that for any Σ and ψ such that $\forall (X : \tau_i) \in \Gamma. \llbracket X \rrbracket_{I, \Sigma} \in \mathbf{T}[\llbracket \tau_i \rrbracket_\psi]$, we have $\llbracket t_1 \rrbracket_{I, \Sigma} \in \mathbf{T}[\llbracket \tau \rrbracket_\psi]$ and $\llbracket t_2 \rrbracket_{I, \Sigma} \in \mathbf{T}[\llbracket \tau \rrbracket_\psi]$. So for these Σ and ψ , we have $\llbracket t_1 = t_2 \rrbracket_\Sigma \in \llbracket \text{bool} \rrbracket_\psi$. Therefore, $\Gamma \Vdash_I t_1 = t_2 : \text{bool}$.

□

The fact that we have a context that gives the type for each variable raises some interesting questions: is there a preferred context for each term? And, given a term t is there a typing representing all possible typings of t ? An answer to these questions is related with the notion of *principal typing* Jim (1996), which we define here appropriate to our system:

Definition 1: A *principal typing* is a pair (Γ, τ) , such that $\Gamma, \Delta \vdash t : \tau$ and for every other pair (Γ', τ') such that $\Gamma', \Delta \vdash t : \tau'$, there is a type substitution μ such that $\mu(\Gamma) = \Gamma'$ and $\mu(\tau) = \tau'$.

Note that it is not always the case that the context in a principal typing is a generic context.

Example 4: Let $t = [X \mid Y]$. A principal typing for t is $(\{X : \alpha, Y : \text{list}(\alpha)\}, \text{list}(\alpha))$. Note that any renaming of type variable α defines another principal typing, because principal typings are unique up to renaming of type variables. Also note that the type for Y cannot be a type variable, this, in this example, the context is not generic.

6 Constraints

While this type system specification has great appeal, the question arises as to whether it can be effectively type checked. Indeed it can by an easy recursive implementation of the type rules. We desire even more flexibility than the language provides as it is. To do so, we generalize the problem beyond purely elided type annotations. To check implicit types during unification, we must deduce types that are not present in equality equations. To represent this problem in a broader context, we introduce the notion of type constraint which we add to the usual term unification problem.

There are two kinds of constraints. Equality constraints between terms $t_1 = t_2$, and equality between types $\tau_1 \doteq \tau_2$. We are here using the same symbol for equality constraints and the equality predicate. We argue that the uses are clear from the context.

We say that a set of equality constraints is in normal form if all constraints are of the form $X_i = t_i$, for some term t_i , and there is no other occurrence of any X_i that is on the left-hand side of a constraint anywhere else in the set.

A set of equality constraints in normal form can be interpreted as a substitution. We can interpret every constraint of the form $X_i = t_i$ as a substitution of the form $[X_i \mapsto t_i]$.

A set of type equality constraints is in normal form if all constraints are of the form $\alpha_i \doteq \tau_i$, for some type τ_i , and there is no other occurrence of any α_i on the left-hand side of a constraint anywhere else in the set.

A set of type equality constraints in normal form can be interpreted as a type substitution. We can interpret every constraint of the form $\alpha_i \doteq \tau_i$ as a type substitution of the form $[\alpha_i \mapsto \tau_i]$.

Definition 2: A substitution θ (or type substitution μ) is called a *unifier* for terms t_1 and t_2 (or types τ_1 and τ_2), iff $\theta(t_1) \equiv \theta(t_2)$ (or $\mu(\tau_1) \equiv \mu(\tau_2)$). Terms t_1 and t_2 (or types τ_1 and τ_2) are *unifiable* iff there exists a unifier for them.

Our constraints are supposed to represent equality, either of terms or types. However, in the semantics, we need states and valuations to interpret non-ground terms and types, respectively. We need a way to interpret the constraints semantically, so we define the following.

Definition 3: Let c be a constraint, Σ a state, and ψ a valuation. We say that Σ and ψ model c , and represent it by $\Sigma, \psi \models c$ if:

- c is an equality constraint of the form $t_1 = t_2$, then $\Sigma(t_1) \equiv \Sigma(t_2)$;
- c is a type equality constraint of the form $\tau_1 \doteq \tau_2$, then $\psi(\tau_1) \equiv \psi(\tau_2)$;

We can now extend this definition to a set of constraints.

Definition 4: Let C be a set of equality constraints and S be a set of type equality constraints. We say that a state Σ and a valuation ψ model the pair (C, S) , and represent it by $\Sigma, \psi \models C, S$ iff Σ and ψ model all constraints in both sets.

We now provide an auxiliary definition that relates substitutions and states and use this definition to extend our notion of constraint modelling.

Definition 5: We say that a state Σ follows a substitution θ and represent it by $\Sigma \sim \theta$ iff for any term t , $\llbracket t \rrbracket_{I, \Sigma} = v$ and $\llbracket \mu(t) \rrbracket_{I, \Sigma} = v$. Similarly, a valuation ψ follows a substitution for types μ ($\psi \sim \mu$) iff for any type τ , $\mathbf{T} \llbracket \tau \rrbracket_\psi = \mathbf{T} \llbracket \mu(\tau) \rrbracket_\psi$.

Definition 6: Let C be a set of equality constraints and S be a set of type equality constraints. We say that a substitution θ and a type substitution μ model the pair (C, S) , and represent it by $\theta, \mu \models C, S$, iff for every state Σ and valuation ψ we have that $\Sigma \sim \theta \wedge \psi \sim \mu \implies \Sigma, \psi \models C, S$.

7 Typed Unification Algorithm

The typed unification algorithm performs unification on both the terms given as input and the types for those terms. The intuition is that if the types do not unify, then there is a type error. We will prove this condition in the next section. We follow the approach of Wand (1987): generate constraints for typability and solve them.

7.1 Constraint Generation

Guided by the definition of our type system we now define a *constraint typing judgment*, which indicates what constraints must hold for a particular type term-and-context pair to be typable. Let Γ be a generic context. We use the following rules to generate constraints for the unification of two terms t_1 and t_2 . The generated constraints will be the pair (C, T) in $\Gamma \vdash t_1 = t_2 : \text{bool} \mid C \mid T$. In the rules in Figure 2, $\vec{\beta}$ represents a sequence of fresh type variables of the same size of $\vec{\alpha}$ in the corresponding case.

$$\begin{array}{c}
 \text{GVAR} \frac{(X : \alpha) \in \Gamma}{\Gamma, \Delta \vdash X : \alpha \mid \emptyset \mid \emptyset} \qquad \text{GCST} \frac{(k : \forall \vec{\alpha}. \tau) \in \Delta}{\Gamma, \Delta \vdash k : \tau[\vec{\alpha} \mapsto \vec{\beta}] \mid \emptyset \mid \emptyset} \\
 \\
 \text{GCPL} \frac{\begin{array}{c} (f : \forall \vec{\alpha}. \tau_1 \times \dots \times \tau_n \rightarrow \tau) \in \Delta \\ \Gamma, \Delta \vdash t_1 : \tau_1' \mid \emptyset \mid T_1 \quad \dots \quad \Gamma, \Delta \vdash t_n : \tau_n' \mid \emptyset \mid T_n \end{array}}{\Gamma, \Delta \vdash f(t_1, \dots, t_n) : \tau[\vec{\alpha} \mapsto \vec{\beta}] \mid \emptyset \mid T_1 \cup \dots \cup T_n \cup \{\tau_1 \doteq \tau_1[\vec{\alpha} \mapsto \vec{\beta}], \dots, \tau_n \doteq \tau_n[\vec{\alpha} \mapsto \vec{\beta}]\}} \\
 \\
 \text{GEQU} \frac{\Gamma, \Delta \vdash t_1 : \tau_1 \mid C_1 \mid T_1 \quad \Gamma, \Delta \vdash t_2 : \tau_2 \mid C_2 \mid T_2}{\Gamma, \Delta \vdash t_1 = t_2 : \text{bool} \mid \{t_1 = t_2\} \mid T_1 \cup T_2 \cup \{\tau_1 \doteq \tau_2\}}
 \end{array}$$

Fig. 2. Constraint Typing Judgment

Example 5: Let Γ be a generic context (we will denote the type variable associated with each variable X by α_X), $\Delta = \{1 : \text{int}, [] : \forall\alpha.\text{list}(\alpha), \text{cons} : \forall\beta.\beta \times \text{list}(\beta) \rightarrow \text{list}(\beta)\}$, $C = \{\text{cons}(X, []) = \text{cons}(1, Y)\}$, and $\Lambda = (\text{cons} : \forall\beta.\beta \times \text{list}(\beta) \rightarrow \text{list}(\beta)) \in \Delta$. The following constraint type judgements hold:

$$\frac{\frac{(X : \alpha_X) \in \Gamma}{\Gamma, \Delta \vdash X : \alpha_X \mid \emptyset \mid \emptyset} \quad \frac{([], \forall\alpha.\text{list}(\alpha)) \in \Delta}{\Gamma, \Delta \vdash [] : \text{list}(\gamma) \mid \emptyset \mid \emptyset} \quad \Lambda}{\Gamma, \Delta \vdash \text{cons}(X, []) : \text{list}(\nu) \mid \emptyset \mid \{\alpha_X = \nu, \text{list}(\gamma) = \text{list}(\nu)\} (= T_1)}$$

$$\frac{\frac{(1 : \text{int}) \in \Delta}{\Gamma, \Delta \vdash 1 : \text{int} \mid \emptyset \mid \emptyset} \quad \frac{(Y : \alpha_Y) \in \Gamma}{\Gamma, \Delta \vdash Y : \alpha_Y \mid \emptyset \mid \emptyset} \quad \Lambda}{\Gamma, \Delta \vdash \text{cons}(1, Y) : \text{list}(\eta) \mid \emptyset \mid \{\text{int} = \eta, \alpha_Y = \text{list}(\eta)\} (= T_2)}$$

$$\frac{\Gamma, \Delta \vdash \text{cons}(1, Y) : \text{list}(\eta) \mid \emptyset \mid T_1 \quad \Gamma, \Delta \vdash \text{cons}(X, []) : \text{list}(\nu) \mid \emptyset \mid T_2}{\Gamma, \Delta \vdash \text{cons}(X, []) = \text{cons}(1, Y) : \text{bool} \mid C \mid T_1 \cup T_2 \cup \{\text{list}(\nu) = \text{list}(\eta)\}}$$

We will now prove that constraint generation is sound, i.e., if we generate constraints (both equality and type equality constraints), any model for them applied to Γ and the type τ is derivable in the syntactic typing system.

Theorem 2 - Soundness of the Constraint Generation: If $\Gamma, \Delta \vdash t : \tau \mid C \mid T$ and $\mu \models T$, then $\mu(\Gamma), \Delta \vdash t : \mu(\tau)$ is derivable in the type system.

Proof

We will prove this theorem by induction on the derivation.

- If t is a variable X , then we have $\Gamma, \Delta \vdash X : \alpha \mid \emptyset \mid \emptyset$. Any type substitution μ is such that $\mu \models \emptyset$. And, for any μ , since $\mu(\alpha)$ will be the same in Γ and in the consequent of the rule, $\mu(\Gamma), \Delta \vdash X : \mu(\alpha)$ is derivable in the syntactic typing system by a single application of rule VAR.
- If t is a constant k , then we have $\Gamma, \Delta \vdash k : \tau[\vec{\alpha} \mapsto \vec{\beta}] \mid \emptyset \mid \emptyset$, where $(k : \forall\vec{\alpha}.\tau) \in \Delta$. Any type substitution μ is such that $\mu \models \emptyset$. Then, for any such μ we can have the derivation in the syntactic system using a single application of rule CST, using $\mu(\tau[\vec{\alpha} \mapsto \vec{\beta}]) = \tau[\vec{\alpha} \mapsto \mu(\vec{\beta})]$.
- If t is a complex term $f(t_1, \dots, t_n)$, then we have $\Gamma, \Delta \vdash f(t_1, \dots, t_n) : \tau[\vec{\alpha} \mapsto \vec{\beta}] \mid \emptyset \mid T_1 \cup \dots \cup T_n \cup \{\tau_{t_1} \doteq \tau_1[\vec{\alpha} \mapsto \vec{\beta}], \dots, \tau_{t_n} \doteq \tau_n[\vec{\alpha} \mapsto \vec{\beta}]\}$, given $\Gamma, \Delta \vdash t_i : \tau_i \mid \emptyset \mid T_i$, for $i = 1, \dots, n$. We also know that $\mu \models T_1 \cup \dots \cup T_n \cup \{\tau_{t_1} \doteq \tau_1[\vec{\alpha} \mapsto \vec{\beta}], \dots, \tau_{t_n} \doteq \tau_n[\vec{\alpha} \mapsto \vec{\beta}]\}$, and any such μ is such that $\mu \models T_i$ and $\mu \models \tau_{t_i} \doteq \tau_i[\vec{\alpha} \mapsto \vec{\beta}]$, for each $i = 1, \dots, n$. By the induction hypothesis, we have $\mu(\Gamma), \Delta \vdash t_i : \mu(\tau_{t_i})$. But we know that $\mu(\tau_{t_i}) = \mu(\tau_i[\vec{\alpha} \mapsto \vec{\beta}])$, since $\mu \models \tau_{t_i} \doteq \tau_i[\vec{\alpha} \mapsto \vec{\beta}]$, for all $i = 1, \dots, n$. So we also have $\mu(\Gamma), \Delta \vdash t_i : \mu(\tau_i[\vec{\alpha} \mapsto \vec{\beta}])$. Therefore, by a single application of the CPL rule, we get $\mu(\Gamma), \Delta \vdash f(t_1, \dots, t_n) : \mu(\tau[\vec{\alpha} \mapsto \vec{\beta}])$.
- If t is an equality $t_1 = t_2$, then we have $\Gamma, \Delta \vdash t_1 = t_2 : \text{bool} \mid \{t_1 = t_2\} \mid T_1 \cup T_2 \cup \{\tau_1 \doteq \tau_2\}$, given $\Gamma, \Delta \vdash t_1 : \tau_1 \mid \emptyset \mid T_1$ and $\Gamma, \Delta \vdash t_2 : \tau_2 \mid \emptyset \mid T_2$. We also know that

$\mu \models T_1 \cup T_2 \cup \{\tau_1 \doteq \tau_2\}$, and any such μ is such that $\mu \models T_1$, $\mu \models T_2$, and $\mu \models \tau_1 \doteq \tau_2$. By the induction hypothesis, we have $\mu(\Gamma), \Delta \vdash t_1 : \mu(\tau_1)$ and $\mu(\Gamma), \Delta \vdash t_2 : \mu(\tau_2)$, but since $\mu \models \tau_1 \doteq \tau_2$, we know that $\mu(\tau_1) \equiv \mu(\tau_2)$. So by a single application of rule EQU, we get $\mu(\Gamma), \Delta \vdash t_1 = t_2 : \mu(bool)$, and $\mu(bool) = bool$.

□

7.2 Constraint Solving

In this section we present a procedure that generalizes unification Robinson (1965) to account for type constraints and produces solutions, where possible. Since each rule simplifies the constraints, together they induce a straightforward decision procedure for type and term constraints.

Suppose we want to unify two terms t_1 and t_2 . Let us have $\Gamma \vdash t_1 = t_2 : bool \mid C \mid T$ derived in the constraint generation step. Then we apply the following rewriting rules to the tuple (C, T) , until none applies. 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. $(C, \{f(\tau_1, \dots, \tau_n) \doteq f(\tau'_1, \dots, \tau'_n)\} \cup Rest) \rightarrow (C, \{\tau_1 \doteq \tau'_1, \dots, \tau_n \doteq \tau'_n\} \cup Rest)$
2. $(C, \{\tau \doteq \tau\} \cup Rest) \rightarrow (C, Rest)$
3. $(C, \{f(\tau_1, \dots, \tau_n) \doteq g(\tau'_1, \dots, \tau'_m)\} \cup Rest) \rightarrow wrong$, if $f \neq g$ or $n \neq m$
4. $(C, \{\tau \doteq \alpha\} \cup Rest) \rightarrow (C, \{\alpha \doteq \tau\} \cup Rest)$, τ is not a type variable
5. $(C, \{\alpha \doteq \tau\} \cup Rest) \rightarrow (C, \{\alpha \doteq \tau\} \cup Rest[\alpha \mapsto \tau])$, if α does not occur in τ
6. $(C, \{\alpha \doteq \tau\} \cup Rest) \rightarrow wrong$, if α occurs in τ
7. $(\{f(t_1, \dots, t_n) = f(s_1, \dots, s_n)\} \cup Rest, T) \rightarrow (\{t_1 = s_1, \dots, t_n = s_n\} \cup Rest, T)$
8. $(\{t \doteq t\} \cup Rest, T) \rightarrow (Rest, T)$
9. $(\{f(t_1, \dots, t_n) = g(s_1, \dots, s_m)\} \cup Rest, T) \rightarrow false$, if $f \neq g$ or $n \neq m$
10. $(\{t \doteq X\} \cup Rest, T) \rightarrow (\{X \doteq t\} \cup Rest, T)$, t is not a variable
11. $(\{X \doteq t\} \cup Rest, T) \rightarrow (\{X \doteq t\} \cup Rest[X \mapsto t], T)$, if X does not occur in t
12. $(\{X \doteq t\} \cup Rest, T) \rightarrow false$, if X occurs in t .

Example 6: Let $C = \{cons(X, []) = cons(1, Y)\}$ and $T = \{\alpha_X = \nu, list(\gamma) = list(\nu), int = \eta, \alpha_Y = list(\eta), list(\nu) = list(\eta)\}$. Step-by-step the algorithm rewrite the pair (C, T) as follows:

$$\begin{aligned}
(C, T) &\rightarrow (C, \{\alpha_X = \nu, \gamma = \nu, int = \eta, \alpha_Y = list(\eta), list(\nu) = list(\eta)\}) \rightarrow \\
&(C, \{\alpha_X = \nu, \gamma = \nu, int = \eta, \alpha_Y = list(\eta), \nu = \eta\}) \rightarrow \\
&(C, \{\alpha_X = \nu, \gamma = \nu, \eta = int, \alpha_Y = list(\eta), \nu = \eta\}) \rightarrow \\
&(C, \{\alpha_X = \nu, \gamma = \nu, \eta = int, \alpha_Y = list(int), \nu = int\}) \rightarrow \\
&(C, \{\alpha_X = \nu, \gamma = int, \eta = int, \alpha_Y = list(int), \nu = int\} (= T')) \rightarrow \\
&(\{X = 1, [] = Y\}, T') \rightarrow (\{X = 1, Y = []\}, T')
\end{aligned}$$

Note that, in the final pair, no more rules apply and we can interpret this pair as a pair of substitutions for terms and for types, respectively.

7.3 Properties of the Regular Typed Unification Algorithm

There are several important properties of our algorithm. Firstly, it always terminates. Secondly, it is correct, meaning that the result is the same as we would have gotten in the equality theory defined for $=$ semantically. One big obstacle for this second property is that terms may not be ground when we want to unify them, and semantically we always need a state to evaluate variables. We will be conservative and assume that if there is a possible state for which the terms have values in the same semantic domain, then there is no type error (yet). Similarly, if there is a state for which the terms have the same semantic value, then the result is not *false* (yet).

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

Theorem 3 - Termination: Let (C, T) be the pair of sets of constraints generated for terms t_1 and t_2 . The rewrite system always terminates, returning a pair of unifiers, false, or wrong.

Proof

We divide the algorithm in two parts. The first consists of the rules 1 to 6, and the second of the rules 7 to 12. Each of these parts are the Martelli-Montanari algorithm Martelli and Montanari (1982) for its corresponding kind of constraints, type equality and equality, respectively. Therefore they terminate.

For a formal proof for the termination of the Martelli-Montanari algorithm, we defer the reader to Martelli and Montanari (1982).

Moreover, if the Martelli-Montanari terminates, the output is either a most general unifier, or the algorithm fails. In the first part, failure is represented by *wrong*, and in the second part, it is represented by *false*. So our algorithm either terminates and outputs *wrong*, *false*, or both parts succeed and the algorithm outputs a pair of most general unifiers. \square

We now know that the algorithm terminates, and what the outputs might be. We will additionally prove that the result is semantically valid. We start by proving a few auxiliary lemmas.

Lemma 1 - Rewriting Consistency: Let $(C, T) \rightarrow (C', T')$ be a step in the typed unification algorithm, such that the output is not *false* nor *wrong*. Then, if for all equality constraints $(t_1 = t_2) \in C'$ the substitution θ is a unifier of t_1 and t_2 , then θ is also a unifier of each equality constraint in C . Same applies to T' and T , with a type substitution μ .

Proof

We will prove this by case analysis.

1. C' and C are equal so, trivially, any substitution θ that unifies each equality constraint in C also unifies each equality constraint in C' . Now suppose that μ is a type substitution such that $\mu(\tau_i) = \mu(\tau'_i)$, for $i = 1, \dots, n$, then, also $\mu(f(\tau_1, \dots, \tau_n)) = \mu(f(\tau'_1, \dots, \tau'_n))$.

All other type equality constraints in T are also in T' , so any unifier of T' is a unifier of T .

2. C' and C are equal so, trivially, any substitution θ that unifies each equality constraint in C also unifies each equality constraint in C' . All type equality constraints in T' are also in T , so all unifiers of T' are unifiers of that subset of T . Moreover, T has one more type equality constraint $\tau \doteq \tau$, but any substitution, in particular any unifier of T' is also a unifier of τ with itself.
3. This case does not apply, since the output is *wrong*.
4. C' and C are equal so, trivially, any substitution θ that unifies each equality constraint in C also unifies each equality constraint in C' . Any unifier of T' is also a unifier of T , since swapping the terms on a type equality constraints does not change the fact that a substitution is a unifier.
5. C' and C are equal so, trivially, any substitution θ that unifies each equality constraint in C also unifies each equality constraint in C' . Suppose μ is a unifier of T' , then $\mu(\alpha) = \mu(\tau)$. Therefore, since $T' = T[\alpha \mapsto \tau]$ for all constraints except $\alpha = \tau$, then $\mu(T') = \mu(T[\alpha \mapsto \tau]) = (\mu \circ [\alpha \mapsto \tau])(T)$ but since $\mu(\alpha) = \mu(\tau)$, then $(\mu \circ [\alpha \mapsto \tau])(T) = \mu(T)$. So μ is also a unifier of T .
6. This case does not apply, since the output is *wrong*.

The proof for the rest of the cases is similar to the proof for the cases 1 to 6, except we replace type equality constraints with equality constraints, type substitution with substitution, and *wrong* with *false*. \square

Lemma 2 - Self-satisfiability: Suppose C is a set of equality constraints in normal form. Then, C can be interpreted as a substitution θ , and θ is a unifier of all constraints in C . Same can be said for a set of type equality constraints in normal form T .

Proof

If C is in normal form, then $C = \{X_1 = t_1, \dots, X_n = t_n\}$, where X_i is a variable and none of X_i occurs in any t_i . So, when we interpret C as a substitution θ , we will have $\theta = [X_1 \mapsto t_1, \dots, X_n \mapsto t_n]$. When we apply θ to each constraint in C , we will get $\theta(C) = \{\theta(X_1) = \theta(t_1), \dots, \theta(X_n) = \theta(t_n)\}$, but since none of the variables X_i occur in any t_i , then $\theta(t_i) = t_i$. Moreover, $\theta(X_i) = t_i$. So we get $\theta(C) = \{t_1 = t_1, \dots, t_n = t_n\}$. Therefore, θ is a unifier of all constraints in C . The proof for type equality constraints is similar to this one, replacing substitutions with type substitutions and terms with type terms. \square

We are now ready to prove the following theorem that proves the algorithm outputs a semantically correct value.

Theorem 4 - Soundness of the Typed Unification Algorithm: Let t_1 and t_2 be the input to the typed unification algorithm, and $\Gamma \vdash t_1 = t_2 \mid C \mid T$. Suppose $(C, T) \rightarrow^* R$.

1. If $R = (\theta, \mu)$, a pair of substitutions for terms and types respectively, then $\theta, \mu \models C, T$.

2. If $R = \text{false}$, then there is no substitution θ such that $\theta \models C$, but there is a type substitution μ such that $\mu \models T$.
3. If $R = \text{wrong}$, then there is no type substitution μ such that $\mu \models T$.

Proof

The proof for (1) follows from Lemmas 1 and 2. We get that θ and μ are unifiers of C and T , respectively.

The proof for (2) follows from the fact that the Martelli-Montanari algorithm is complete, i.e., if there was a unifier for the equality constraint set C , then it would have been obtained. Therefore there is no unifier for C , so there is no θ such that $\theta \models C$. However, since we got to the second part of the algorithm, then we were able to find a unifier for the type equality constraints. This means that there is at least one unifier for T , so there is a μ such that $\mu \models T$.

The proof for (3) follows from the fact that the Martelli-Montanari algorithm is complete, i.e., if there was a unifier for the type equality constraint set T , then it would have been obtained. Therefore there is no unifier for T , so there is no μ such that $\mu \models T$. \square

Additionally, we want to prove that we always get the principal typing for a term using the constraint generation and solving.

Theorem 5 - Completeness of the Typed Unification Algorithm: Let t be term, or a unification of two terms, Γ be a generic context, and Δ be type declarations for constants and function symbols. If $\Gamma, \Delta \vdash t : \tau \mid C \mid T$ and $(C, T) \rightarrow^* (\theta, \mu)$. Then $(\mu(\Gamma), \mu(\tau))$ is a principal typing of t .

Proof

We will prove by structural induction on t .

- If t is a variable X , then $(X : \alpha) \in \Gamma$. We know that $\Gamma, \Delta \vdash X : \alpha \mid \emptyset \mid \emptyset$ by a single application of GVAR. $(\emptyset, \emptyset) \rightarrow ([], [])$, where $[]$ are each the identity substitution for variables and type variables, respectively. Therefore $[](\alpha) = \alpha$, and any type τ derived in the type system such that $(X : \tau) \in \Gamma'$, then τ is an instance of α .
- If t is a constant k , then $(k : \forall \vec{\alpha}. \tau) \in \Delta$. We know that $\Gamma, \Delta \vdash k : \tau[\vec{\alpha} \mapsto \vec{\sigma}]$, for any $\vec{\sigma}$. We get by a single application of rule GCST that $\Gamma, \Delta \vdash k : \tau[\vec{\alpha} \mapsto \vec{\beta}] \mid \emptyset \mid \emptyset$. $(\emptyset, \emptyset) \rightarrow ([], [])$, where $[]$ are each the identity substitution for variables and type variables, respectively. Therefore $[](\tau[\vec{\alpha} \mapsto \vec{\beta}]) = \tau[\vec{\alpha} \mapsto \vec{\beta}]$, and we know that any $\vec{\sigma}$ is an instance of $\vec{\beta}$.
- If t is a complex term $f(t_1, \dots, t_n)$, then $(f : \forall \vec{\alpha}. \tau_1 \times \dots \times \tau_n \rightarrow \tau) \in \Delta$. By the induction hypothesis, we know that if $(\emptyset, T_i) \rightarrow^* ([], \mu_i)$, then $(\mu_i(\Gamma), \mu_i(\tau_i))$ is a principal typing of t_i . Now suppose $(\emptyset, T_1 \cup \dots \cup T_n \cup \{\tau'_1 \doteq \tau_1[\vec{\alpha} \mapsto \vec{\beta}], \dots, \tau'_n \doteq \tau_n[\vec{\alpha} \mapsto \vec{\beta}]\}) \rightarrow^* ([], \mu)$. We know that $\mu(\tau'_i) = \mu(\tau_i[\vec{\alpha} \mapsto \vec{\beta}])$ for all $i = 1, \dots, n$. So we can derive $\mu(\Gamma), \Delta \vdash t_i : \mu(\tau_i[\vec{\alpha} \mapsto \vec{\beta}])$, and by a single application of rule CPL, we get $\mu(\Gamma), \Delta \vdash f(t_1, \dots, t_n) : \mu(\tau[\vec{\alpha} \mapsto \vec{\beta}])$. Now we need to prove that this typing $(\mu(\Gamma), \mu(\tau[\vec{\alpha} \mapsto \vec{\beta}]))$ is the principal typing. Suppose we had another typing that was not an instance of this one $(\mu'(\Gamma), \mu'(\tau[\vec{\alpha} \mapsto \vec{\beta}]))$. Since μ is an MGU of $T_1 \cup \dots \cup T_n \cup \{\tau'_1 \doteq \tau_1[\vec{\alpha} \mapsto \vec{\beta}], \dots, \tau'_n \doteq \tau_n[\vec{\alpha} \mapsto \vec{\beta}]\}$, then either for some i $\mu' \not\models T_i$, or for some i $\mu' \not\models \tau'_i \doteq \tau_i[\vec{\alpha} \mapsto \vec{\beta}]$. If the former

is true, then $(\mu'(\Gamma), \mu'(\tau(t_i)))$ is not an instance of the principal typing for t_i and therefore cannot be derived in the type system. If the latter is true, then $\mu'(\tau_i[\vec{\alpha} \mapsto \vec{\beta}]) \neq \mu'(\tau(t_i))$ and we cannot use the rule CPL in the type system. Therefore, $(\mu(\Gamma), \mu(\tau[\vec{\alpha} \mapsto \vec{\beta}]))$ is the principal typing for $f(t_1, \dots, t_n)$.

- Suppose t is an equality $t_1 = t_2$. By the induction hypothesis, we know that if $(\emptyset, T_i) \rightarrow^* ([\], \mu_i)$, then $(\mu_i(\Gamma), \mu_i(\tau_i))$ is a principal typing of t_i . Now suppose $(\{t_1 = t_2\}, T_1 \cup T_2 \cup \{\tau_1 \doteq \tau_2\}) \rightarrow^* (\theta, \mu)$. We know that $\mu(\tau_i)$ is an instance of $\mu_i(\tau_i)$, so we can derive $\mu(\Gamma), \Delta \vdash t_i : \mu(\tau_i)$ in the type system. By a single application of rule EQU, we get $\mu(\Gamma), \Delta \vdash t_1 = t_2 : \text{bool}$. So $(\mu(\Gamma), \mu(\text{bool}))$ is a typing. Suppose it was not the principal typing. Suppose we had another typing that was not an instance of this one $(\mu'(\Gamma), \mu'(\text{bool}))$. For all μ , $\mu(\text{bool}) = \text{bool}$. Since μ is an MGU of $T_1 \cup T_2 \cup \{\tau_1 \doteq \tau_2\}$, then if μ' is not an instance either for some i $\mu' \not\models T_i$ or $\mu' \not\models \tau_1 \doteq \tau_2$. If the former is true, then $(\mu'(\Gamma), \mu'(\tau_1))$ is not an instance of its principal typing, so it cannot be derived in the type system. If the latter is true, the types for t_1 and t_2 are different and we cannot apply rule EQU, so we could not have this derivation in the type system. Therefore, $(\mu(\Gamma), \mu(\text{bool}))$ is the principal typing for $t_1 = t_2$.

□

8 Using Regular Typed Unification in Practice

Our regular typed unification provides some foundation for the use of regular types to dynamically catch erroneous Prolog behaviors. Indeed, one of the original motivations for this work was to understand how to extend the YAP Prolog system Costa et al. (2012) with an effective dynamic typing. Here we will see that, due to efficiency matters, this extension is not able to replace first order unification by regular typed unification in the Prolog engine.

In Barbosa et al. (2022b) we proposed a typed SLD-resolution (TSLD) which used our previous notion of typed unification. Our goal now is to effectively extend SLD resolution with unification of terms typed by regular types.

A TSLD-tree branch may result in *true*, *false*, or *wrong*, depending on the same results for the unifications in the branch. In Barbosa et al. (2022b), each TSLD-tree branch that eventually outputs *false*, needed to continue execution on the same branch in order to check if there was a type error in some other atom in the query. This lead to a drastic increase in the runtime of programs.

Example 7: Consider the following (unrealistic) but possible program:

`p(0).`

and query: `?- p(1), ..., p(3000), p(a)`. In Prolog SLD-resolution, the first atom in the query fails to unify with the only clause in the program for predicate *p*, which immediately fails. So there is a single SLD-step.

In the TSLD-resolution defined in Barbosa et al. (2022b), since the first 3000 queries return *false*, we need to reach step 3001 in order to obtain the value *wrong*. Note that if the query was `?- p(1), ..., p(3000)` we would only take a single step less, but get the same result as SLD-resolution.

Pragmatically, it is quite impractical to increase the runtime of programs like this in these cases. This reduction on efficiency would make Prolog an unusable programming language for real applications.

Thus, when adding regular typed unification to Prolog we have a compromise between completeness and efficiency.

If the evaluation of a query is *false* we stop execution, and the same happens for *wrong*. However, if the result is *false* and there are other atoms in the query, we cannot assure that the value for that branch is indeed *false*, only that it is not *true*. Thus, in our extension to Prolog we output **no**(?) in these cases. On the other hand, we output **no**(false) if there are no other atoms in the query, and **no**(wrong) if the branch ends on *wrong*.

Example 8: Consider the same program as above and the query `?- p(2),p(a)`. After a single TSLD-step we get *false*, but there is another atom in the query, so Prolog now outputs **no**(?).

If the query, however, is `?- p(a),p(2)` we would output **no**(wrong).

One can ask what would be the point then, to sometimes detect type errors but other times fail to detect them. Well, in many programs, for some queries, we are *always* able to detect the type error.

Example 9: Consider the predicate that calculates the length of a list:

```
length([], 0).
length([_|T],N) :- length(T,N1), N is N + 1.
```

One typical bug in Prolog, is to swap the arguments of a predicate. Now note that, in this case, if we have the erroneous query `?- length(3,[a,b,c])`, both branches of the TSLD-tree output *wrong* since there is a type error in the first argument (and also in the second).

9 Conclusion and Future Work

In this paper we present a new unification algorithm for typed terms where types are a subset of regular types which correspond exactly to the usual notion of abstract data types. The work presented here inspires the following possible tasks: 1) Investigate the extension of regular typed unification to other forms of unification. In particular, investigate the extensions to higher-order unification and constraint solving in specific constraint domains. 2) Develop a meta-theory of regular typed unification. In particular, investigate conditions under which unification of terms typed by extensions of regular types, such as closed types or dependent types, satisfies the most general and principal types property and is decidable. 3) Investigate the complexity of regular typed unification. Separately, also investigate the complexity of the set of programs that we are able to type by restrictions of regular types.

Acknowledgements This work was partially financially supported by

UIDB/00027/2020 of the Artificial Intelligence and Computer Science Laboratory, LIACC, funded by national funds through the FCT/MCTES (PIDDAC).

References

- APT, K. R. 1996. *From Logic Programming to Prolog*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- BARBOSA, J., FLORIDO, M., AND SANTOS COSTA, V. Data type inference for logic programming. In DE ANGELIS, E. AND VANHOOF, W., editors, *Logic-Based Program Synthesis and Transformation 2022a*, pp. 16–37, Cham. Springer International Publishing.
- BARBOSA, J., FLORIDO, M., AND SANTOS COSTA, V. Typed SLD-Resolution: Dynamic typing for logic programming. In VILLANUEVA, A., editor, *Logic-Based Program Synthesis and Transformation 2022b*, pp. 123–141, Cham. Springer International Publishing.
- CODISH, M. AND LAGOON, V. 2000. Type dependencies for logic programs using aci-unification. *Theor. Comput. Sci.*, 238, 1-2, 131–159.
- COSTA, V. S., ROCHA, R., AND DAMAS, L. 2012. The YAP Prolog System. *Theory and Practice of Logic Programming*, 12, 1-2, 5–34.
- DAMAS, L. AND MILNER, R. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages 1982*, pp. 207–212.
- DART, P. W. AND ZOBEL, J. A regular type language for logic programs. In PFENNING, F., editor, *Types in Logic Programming 1992*, pp. 157–187. The MIT Press.
- FRÜHWIRTH, T. W., SHAPIRO, E. Y., VARDI, M. Y., AND YARDENI, E. Logic programs as types for logic programs. In *Proc. of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Netherlands, 1991* 1991, pp. 300–309.
- GALLAGHER, J. P. AND HENRIKSEN, K. S. Abstract Domains Based on Regular Types. In *International Conference on Logic Programming 2004*, pp. 27–42. Springer.
- HERMENEGILDO, M. V., MORALES, J. F., LÓPEZ-GARCÍA, P., AND CARRO, M. Types, modes and so much more - the prolog way. In *Prolog: The Next 50 Years 2023*, volume 13900 of *Lecture Notes in Computer Science*, pp. 23–37. Springer.
- JIM, T. What are principal typings and what are they good for? In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996* 1996, pp. 42–53. ACM Press.
- LLOYD, J. W. 1984. *Foundations of Logic Programming*. Springer-Verlag, Berlin, Heidelberg.
- MARTELLI, A. AND MONTANARI, U. 1982. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4, 2, 258–282.
- MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17, 3, 348–375.
- MISHRA, P. 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* 1984, pp. 289–298. IEEE-CS.
- ROBINSON, J. A. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12, 1, 23–41.
- SCHRIJVERS, T., BRUYNOOGHE, M., AND GALLAGHER, J. P. From monomorphic to polymorphic well-typings and beyond. In *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Valencia, Spain, July 17-18, 2008, Revised Selected Papers 2008a*, volume 5438 of *Lecture Notes in Computer Science*, pp. 152–167. Springer.
- SCHRIJVERS, T., COSTA, V. S., WIELEMAKER, J., AND DEMOEN, B. Towards Typed Prolog.

- In *Logic Programming, 24th International Conference, ICLP 2008b*, volume 5366 of *Lecture Notes in Computer Science*, pp. 693–697. Springer.
- SMOLKA, G. Logic programming with polymorphically order-sorted types. In *Algebraic and Logic Programming, International Workshop, Gaussig, GDR, November 14-18, 1988, Proceedings 1988*, volume 343 of *Lecture Notes in Computer Science*, pp. 53–70. Springer.
- WALTHER, C. 1988. Many-sorted unification. *J. ACM*, 35, 1, 1–17.
- WAND, M. 1987. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, 10, 115–122.
- YARDENI, E. AND SHAPIRO, E. 1991. A type system for logic programs. *The Journal of Logic Programming*, 10, 2, 125–153.
- ZOBEL, J. Derivation of polymorphic types for Prolog programs. In *Logic Programming, Proceedings of the Fourth International Conference, Melbourne, 1987* 1987.