A Type System for Tom

Claude Kirchner, Pierre-Etienne Moreau, Cláudia Tavares

A Type System for Tom

Claude Kirchner
INRIA
Centre de Recherche
INRIA Bordeaux - Sud-Ouest
351, cours de la Libération,
33405 Talence Cedex France
Claude.Kirchner@inria.fr

Pierre-Etienne Moreau
Cláudia Tavares*
INRIA & LORIA
615 rue du Jardin Botanique, CS 20101
54603 Villers-lès-Nancy Cedex France
Pierre-Etienne.Moreau@loria.fr
Claudia.Tavares@loria.fr

Extending a given language with new dedicated features is a general and quite used approach to make the programming language more adapted to problems. Being closer to the application, this leads to less programming flaws and easier maintenance. But of course one would still like to perform program analysis on these kinds of extended languages, in particular type checking and inference. In this case one has to make the typing of the extended features compatible with the ones in the starting language.
The Tom programming language is a typical example of such a situation as it consists of an extension of Java that adds pattern matching, more particularly associative pattern matching, and reduction strategies.
This paper presents a type system with subtyping for Tom, that is compatible with Java's type system, and that performs both type checking and type inference. We propose an algorithm that checks if all patterns of a Tom program are well-typed. In addition, we propose an algorithm based on equality and subtyping constraints that infers types of variables occurring in a pattern. Both algorithms are exemplified and the proposed type system is showed to be sound and complete.

1 Introduction of the problem: static typing in Tom

We consider here the Tom language, which is an extension of Java that provides rule based constructs. In particular, any Java program is a Tom program. We call this kind of extension *formal islands* [4, 3] where the *ocean* consists of Java code and the *island* of algebraic patterns. For simplicity, we consider here only two new Tom constructs: a `%match` construct and a ‘ (backquote) construct.

The semantics of `%match` is close to the *match* that exists in functional programming languages, but in an imperative context. A `%match` is parameterized by a list of subjects (*i.e.* expressions evaluated to ground terms) and contains a list of rules. The left-hand side of the rules are patterns built upon constructors and fresh variables, without any linearity restriction. The right-hand side is *not* a term, but a Java statement that is executed when the pattern matches the subject. However, thanks to the backquote construct (‘) a term can be easily built and returned. In a similar way to the standard `switch/case` construct, patterns are evaluated from top to bottom. In contrast to the functional *match*, several actions (*i.e.* right-hand sides) may be fired for a given subject as long as no `return` or `break` instruction is executed. To implement a simple reduction step for each rule, it suffices to encode the left-hand side with a pattern and consider the Java statement that returns the right-hand side.

For example, given the sort `Nat` and the function symbols `suc` and `zero`, addition and comparison of Peano integers may be encoded as follows:

*This work was partially supported by The Capes Foundation, Ministry of Education of Brazil. Cx. postal 365, Brasília DF 70359-970, Brazil.

I. Mackie and A. Martins Moreira (Eds.): Tenth International
Workshop on Rule-Based Programming (RULE 2009)
EPTCS 21, 2010, pp. 51–63, doi:10.4204/EPTCS.21.5

```
public Nat plus(Nat t1, Nat t2) {
 %match(t1,t2) {
   x,zero() -> { return `x; }
   x,suc(y) -> { return `suc(plus(x,y)); }
 }
}
```

```
public boolean greaterThan(Nat t1, Nat t2) {
 %match(t1, t2) {
   x,x           -> { return false; }
   suc(x),zero() -> { return true; }
   zero(),suc(y) -> { return false; }
   suc(x),suc(y) -> { return `greaterThan(x,y); }
 }
}
```

In this combination of an ocean language (in our case Java) and island features (in our case abstract data types and matching), it is still an open question to perform type checking and type inference.

Since we want to allow for type inclusion at the pattern level, the first purpose of this paper is to present an extension of the signature definition mechanism allowing for subtypes. In this context we define *Java-like* types and signatures. Therefore the set of types is the union of Java types and abstract data types (i.e. Tom types) where multiple inheritance and overloading are forbidden. For example, given the sorts $\texttt{Int}^+$, $\texttt{Int}^-$, Int and Zero, the type system accepts the declaration $\texttt{Int}^+ <: \texttt{Int} \wedge \texttt{Int}^- <: \texttt{Int}$ but refuses the declaration $\texttt{Zero} <: \texttt{Int}^+ \wedge \texttt{Zero} <: \texttt{Int}^-$. Moreover, a function symbol suc cannot be overloaded on both sorts $\texttt{Int}^+$ and $\texttt{Int}^-$. In order to handle those issues, we propose an algorithm based on unification of equality constraints [14] and simplification of subtype constraints [8, 1, 16]. It infers the types of the variables that occur in a pattern (x and y in the previous example). Moreover, we also propose an algorithm that checks that the patterns occurring in a Tom program are correctly typed.

Of course typing systems for algebraic terms and for rewriting has a long history. It includes the seminal works done on OBJ, order-sorted algebras [10, 9] and Maude [6]; the works done on feature algebras [2] or on membership constraints [11, 7]; and the works on typing rewriting in higher-order settings like [17] or [5]. Largely inspired from these works, our contribution here focusses on the appropriate type system for pattern-matching, possibly modulo associativity, in a Java environment.

2 Type checking

Given a signature Σ_v, the (simplified) abstract syntax of a Tom program is as follows:

$$\begin{array}{lcl} \mathit{rule} & ::= & \mathit{cond} \longrightarrow \mathit{action} \\ \mathit{cond} & ::= & \mathit{term}_1 \lessdot_{[s]} \mathit{term}_2 \mid \mathit{cond}_1 \wedge \mathit{cond}_2 \\ \mathit{term} & ::= & x \mid f(\mathit{term}_1, \ldots, \mathit{term}_n) \\ \mathit{action} & ::= & (\mathit{term}_1, \ldots, \mathit{term}_n) \end{array}$$

The left-hand side of a rule is a conjunction of matching conditions $\mathit{term}_1 \lessdot_{[s]} \mathit{term}_2$ consisting of a pair of terms and where s denotes a sort. We introduce the set $\mathscr{F}$ of free symbols. Terms are many-sorted terms composed of variables $x \in \mathscr{X}$ and function symbols $f \in \mathscr{F}$. The set of terms is written $\mathscr{T}(\mathscr{F}, \mathscr{X})$. In general, an *action* is a Java statement, but for our purpose it is enough to consider an abstraction consisting of terms $e_1, \ldots, e_n \in \mathscr{T}(\mathscr{F}, \mathscr{X})$ whose instantiations are described by the conditions, and used in the Java statement.

Example 2.1. *The last rule of the* greaterThan *function given above can be represented by the following rule expression:*

$$\mathit{suc}(x) \lessdot_{[\mathbb{N}]} t_1 \wedge \mathit{suc}(y) \lessdot_{[\mathbb{N}]} t_2 \longrightarrow (x,y)$$

In a first step, we define $\mathscr{S}$ as a set of sorts and we consider that a *context* Γ is composed of a set of pairs (variable,sort), and (function symbol,rank):

$$\Gamma ::= \varnothing \mid \Gamma_1 \cup \Gamma_2 \mid x : s \mid f : s_1, \ldots, s_n \to s$$

and context access is defined by the function $\texttt{sortOf}(\Gamma, e) : \Gamma \times \mathscr{T}(\mathscr{F}, \mathscr{X}) \to \mathscr{S}$ which returns the sort of term e in the context Γ:

$$\texttt{sortOf}(\Gamma, x) = s, \text{ if } x : s \in \Gamma \qquad \texttt{sortOf}(\Gamma, f(e_1, \ldots, e_n)) = s, \text{ if } f : s_1, \ldots, s_n \to s \in \Gamma$$

where $x \in \mathscr{X}$ and $f \in \mathscr{F}$.

We denote by $\Gamma(x : s)$ the fact that $x : s$ belongs to Γ. Similarly, $\Gamma(f : s_1, \ldots, s_n \to s)$ means that $f : s_1, \ldots, s_n \to s$ belongs to Γ. In Fig. 1 we give a classical type checking system defined by a set of inference rules. Starting from a context Γ and a rule expression π, we say that π is well-typed if $\pi : wt$ can be derived by applying the inference rules. *wt* is a special sort that corresponds to the well-typedness of a *rule* or a condition *cond*.

$$\frac{}{\Gamma(x : s) \vdash x : s}\ \text{T-VAR} \qquad \frac{\Gamma \vdash e_1 : s_1 \quad \ldots \quad \Gamma \vdash e_n : s_n}{\Gamma(f : s_1, \ldots, s_n \to s) \vdash f(e_1, \ldots, e_n) : s}\ \text{T-FUN}$$

$$\frac{\Gamma \vdash e_1 : s \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash (e_1 \ \prec_{[s]} \ e_2) : wt}\ \text{T-MATCH} \qquad \frac{\Gamma \vdash (cond_1) : wt \quad \ldots \quad \Gamma \vdash (cond_n) : wt}{\Gamma \vdash (cond_1 \wedge \ldots \wedge cond_n) : wt}\ \text{T-CONJ}$$

$$\frac{\Gamma \vdash (cond) : wt \qquad \Gamma \vdash e_1 : s_1 \quad \ldots \quad \Gamma \vdash e_n : s_n}{\Gamma \vdash (cond \longrightarrow (e_1, \ldots, e_n)) : wt}\ \text{T-RULE}$$
$$\text{if } \texttt{sortOf}(\Gamma, e_i) = s_i, \text{ for } i \in [1, n]$$

Figure 1: Simple type checking system.

2.1 Subtypes and associative-matching

In order to introduce subtypes in Tom, we refine $\mathscr{S}$ as the set of sorts, equipped with a partial order $<:$, called *subtyping*. It is a binary relation on $\mathscr{S}$ that satisfies reflexivity, transitivity and antisymmetry. Moreover, since we allow for some symbols to be associative, we introduce the set $\mathscr{F}_v$ of variadic symbols to denote them. Now, the set of terms is written $\mathscr{T}(\mathscr{F} \cup \mathscr{F}_v, \mathscr{X})$ and terms are many-sorted variadic terms composed of variables $x \in \mathscr{X}$ and function symbols $f \in \mathscr{F} \cup \mathscr{F}_v$. In the following, we often write ℓ a variadic operator and call it a *list*.

We extend matching over lists to be associative. Therefore a pattern matches a subject considering equality relation modulo flattening. Lists can be denoted by function symbols $\ell \in \mathscr{F}_v$ or by variables $x \in \mathscr{X}$ annotated by *. Such variables, which we write x^*, are called *star variables*. So we consider in the following many-sorted variadic terms composed of variables $x \in \mathscr{X}$, star variables x^* (where $x \in \mathscr{X}$) and function symbols $f \in \mathscr{F} \cup \mathscr{F}_v$. Moreover, we define that function symbols $\ell \in \mathscr{F}_v$ with variable domain (since they have a variable arity) of sort s_1 and codomain s are written $\ell : {s_1}^* \to s$ while star variables x^* are also sorted and written $x^* : s$.

Since terms built from syntactic and variadic operators can have the same codomain, we cannot distinguish one from the other only by theirs sorts. However, this is necessary to know which typing rule applies. Moreover, an insertion of a term can be treated by two ways: given terms $\ell(e_1), \ell(e_2), \ell_1(e_1) \in \mathscr{T}(\mathscr{F} \cup \mathscr{F}_v, \mathscr{X})$ where $\ell, \ell_1 \in \mathscr{F}_v$, we have: 1) an insertion of a list $\ell(e_1)$ into a list $\ell(e_2)$ corresponds

to a concatenation of these both lists resulting in $\ell(e_1,e_2)$; 2) an insertion of a list $\ell_1(e_1)$ into a list $\ell(e_2)$ results in $\ell(\ell_1(e_1),e_2)$. For that reason, it is important to distinguish the list from the inserted term by its function symbol in order to define which typing rule concerned for list must be applied. For this purpose, we introduce a notion of sorts decorated with function symbols, called *types*, to classify terms. The special symbol ? is used as decoration when it is not useful to know what the function symbol is, i.e. when the expected type is known but not the expected function symbol. This leads to a new set of decorated sorts $\mathscr{D}$ which is equipped with a partial order $<:_s$. It is a binary relation on $\mathscr{D}$ where $s_1^{g_1} <:_s s_2^{g_2}$ is equivalent to $s_1 <: s_2 \wedge (g_1 = g_2 \vee g_2 = ?)$.

As pointed out in the introduction, we assume in all that paper that the signatures considered do not have multiple inheritance and that we do not allow function symbol overloading.

Given these notions, we refine the notion of context Γ as a set of subtyping declarations (type,type) and pairs (variable,type), and (function symbol,rank). This is expressed by the following grammar:

$$\Gamma ::= \varnothing \mid \Gamma_1 \cup \Gamma_2 \mid s_1^? \overset{*}{<:_s} s_2^? \mid x : s^g \mid x^* : s^\ell \mid f : s_1^?, \ldots, s_n^? \to s^f \mid \ell : (s_1^?)^* \to s^\ell$$

where $\overset{*}{<:_s}$ corresponds to the reflexive transitive closure of $<:_s$ and context access is refined by the function $\texttt{sortOf}(\Gamma,e) : \Gamma \times \mathscr{T}(\mathscr{F} \cup \mathscr{F}_v, \mathscr{X}) \to \mathscr{D}$ which returns the type of term e in the context Γ:

$$\begin{array}{rclcrcl} \texttt{sortOf}(\Gamma,x) & = & s^g, \text{ if } x : s^g \in \Gamma & \qquad & \texttt{sortOf}(\Gamma, f(e_1,\ldots,e_n)) & = & s^f, \text{ if } f : s_1^?,\ldots,s_n^? \to s^f \in \Gamma \\ \texttt{sortOf}(\Gamma,x^*) & = & s^\ell, \text{ if } x^* : s^\ell \in \Gamma & & \texttt{sortOf}(\Gamma,\ell(e_1,\ldots,e_n,e)) & = & s^\ell, \text{ if } \ell : (s_1^?)^* \to s^\ell \in \Gamma \end{array}$$

where $x \in \mathscr{X}$, $f \in \mathscr{F}$, $\ell \in \mathscr{F}_v$, $g \in \mathscr{F} \cup \mathscr{F}_V \cup \{?\}$ and $s^?, s^f, s^g, s^\ell \in \mathscr{D}$.

The context has at most one declaration of type or signature per term since overloading is forbidden. This means that for $e \in \mathscr{T}(\mathscr{F} \cup \mathscr{F}_v, \mathscr{X})$ and $s_1^{g_1}, s_2^{g_2}$ (where $g_1, g_2 \in \mathscr{F} \cup \mathscr{F}_v \cup \{?\}$ and $s_1^{g_1}, s_2^{g_2} \in \mathscr{D}$) if $e : s_1^{g_1} \in \Gamma$ and $e : s_2^{g_2} \in \Gamma$ then $s_1^{g_1} = s_2^{g_2}$. We denote by $\Gamma(s_1^{g_1} <:_s s_2^{g_2})$ the fact that $s_1^{g_1} <:_s s_2^{g_2}$ belongs to Γ.

2.2 Type checking algorithm

In Fig. 2 we give a type checking system to many-sorted variadic terms applying associative matching. The rules are standard except for the use of decorated types. The most interesting rules are those that apply to lists. They are three: [T-EMPTY] checks if a empty list has the same type declared in Γ; [T-ELEM] is similar to [T-FUN] but is applied to lists; and [T-MERGE] is applied to a concatenation of two lists of type s^ℓ in Γ, resulting in a new list with same type s^ℓ.

The type checking algorithm reads derivations bottom-up. Since the rule [SUB] can be applied to any kind of term, we consider a strategy where it is applied iff no other typing rule can be applied. In practice, [SUB] will be combined with [T-VAR], [T-FUN] and [T-ELEM] and the type $s_1^?$ which appears in the premise will be defined according to the result of function $\texttt{sortOf}(\Gamma,e)$. The algorithm stops if it reaches the [T-VAR] or [T-SVAR] cases, ensuring that the original expression is well-typed, or if none of the type checking rules can be applied, raising an error.

Example 2.2. *Let* $\Gamma = \{\ell : (\mathbb{Z}^?)^* \to \mathbb{Z}^\ell, one : \to \mathbb{N}^{one}, x^* : \mathbb{Z}^\ell, z^* : \mathbb{Z}^\ell, y : \mathbb{Z}^?, \mathbb{N}^? <:_s \mathbb{Z}^?\}$. *Then the expression* $\ell(x^*,y,z^*) \prec\!\!\prec_{[\mathbb{Z}^?]} \ell(one()) \longrightarrow (y)$ *is well-typed and its deduction tree is given in Fig. 3*

3 Type inference

The type system presented in Section 2 needs rules to control its use in order to find the expected deduction tree of an expression. Without these rules it is possible to find more than one deduction tree for the

$$\frac{}{\Gamma(x:s^g)\vdash x:s^g}\ \text{T-VAR}$$
where $g\in\mathscr{F}\cup\mathscr{F}_v\cup\{?\}$

$$\frac{}{\Gamma(x^*:s^\ell)\vdash x^*:s^\ell}\ \text{T-SVAR}$$

$$\frac{\Gamma\vdash e_1:s_1^?\qquad\ldots\qquad\Gamma\vdash e_n:s_n^?}{\Gamma(f:s_1^?,\ldots,s_n^?\to s^f)\vdash f(e_1,\ldots,e_n):s^f}\ \text{T-FUN}$$

$$\frac{}{\Gamma(\ell:(s_1^?)^*\to s^\ell)\vdash \ell():s^\ell}\ \text{T-EMPTY}$$

$$\frac{\Gamma\vdash \ell(e_1,\ldots,e_n):s^\ell\qquad\Gamma\vdash e:s_1^?}{\Gamma(\ell:(s_1^?)^*\to s^\ell)\vdash \ell(e_1,\ldots,e_n,e):s^\ell}\ \text{T-ELEM}$$
if $\texttt{sortOf}(\Gamma,e)\neq s^\ell$ and $e\neq x^*$

$$\frac{\Gamma\vdash \ell(e_1,\ldots,e_n):s^\ell\qquad\Gamma\vdash e:s^\ell}{\Gamma(\ell:(s_1^?)^*\to s^\ell)\vdash \ell(e_1,\ldots,e_n,e):s^\ell}\ \text{T-MERGE}$$
if $\texttt{sortOf}(\Gamma,e)=s^\ell$

$$\frac{\Gamma\vdash e:s_1^{g_1}}{\Gamma(s_1^{g_1}<:_s s^g)\vdash e:s^g}\ \text{SUB}$$
where $g,g_1\in\mathscr{F}\cup\mathscr{F}_v\cup\{?\}$

$$\frac{\Gamma\vdash e:s^h}{\Gamma\vdash e:s^?}\ \text{GEN}$$
if $\texttt{sortOf}(\Gamma,e)=s^h$, where $h\in\mathscr{F}\cup\mathscr{F}_v$

$$\frac{\Gamma\vdash e_1:s^?\qquad\Gamma\vdash e_2:s^?}{\Gamma\vdash(e_1\ \prec_{[s^?]}\ e_2):wt}\ \text{T-MATCH}$$

$$\frac{\Gamma\vdash(cond_1):wt\qquad\ldots\qquad\Gamma\vdash(cond_n):wt}{\Gamma\vdash(cond_1\wedge\ldots\wedge cond_n):wt}\ \text{T-CONJ}$$

$$\frac{\Gamma\vdash(cond):wt\qquad\Gamma\vdash e_1:s_1^{g_1}\qquad\ldots\qquad\Gamma\vdash e_n:s_n^{g_n}}{\Gamma\vdash(cond\longrightarrow(e_1,\ldots,e_n)):wt}\ \text{T-RULE}$$
if $\texttt{sortOf}(\Gamma,e_i)=s_i^{g_i}$, where $g_i\in\mathscr{F}\cup\mathscr{F}_v\cup\{?\}$ for $i\in[1,n]$

Figure 2: Type checking rules.

same expression. For instance, in Example 2.2, the rule [SUB] can be applied to the leaves resulting of application of rule [T-VAR]. The resulting tree will still be a valid deduction tree since the variables in the leaves will have type $\mathbb{N}^?$ instead of type $\mathbb{Z}^?$ declared in the context and $\mathbb{N}^?<:_s\mathbb{Z}^?$. For that reason, we are interested in defining another type system able to infer the most general types of terms. We add type variables in the set of types (defined up to here as a set of decorated sorts) to describe a possibly infinite set of decorated sorts. The set of types $\mathscr{T}_{ype}(\mathscr{D}\cup\{wt\},\mathscr{V})$ is given by a set of decorated sorts $\mathscr{D}$, a set of type variables $\mathscr{V}$ and a special sort wt:

$$\tau ::= \alpha \mid s^g \mid wt$$

where $\tau\in\mathscr{T}_{ype}(\mathscr{D}\cup\{wt\},\mathscr{V})$, $\alpha\in\mathscr{V}$, $g\in\mathscr{F}\cup\mathscr{F}_v\cup\{?\}$ and $s^g\in\mathscr{D}$.

In order to build the subtyping rule into the rules, we use a *constraint set* C to store all equality and subtyping constraints. These constraints limit types that terms can have. The language $\mathscr{C}$ is built from the set of types $\mathscr{T}_{ype}(\mathscr{D}\cup\{wt\},\mathscr{V})$ and the operators "$=_s$" (equality) and "$<:_s$" (extension to $\mathscr{T}_{ype}(\mathscr{D}\cup\{wt\},\mathscr{V})$ of the partial order defined in Subsection 2.1):

$$c ::= \tau_1 =_s \tau_2 \mid \tau_1 <:_s \tau_2$$

where $c\in\mathscr{C}$, $\tau_1,\tau_2\in\mathscr{T}_{ype}(\mathscr{D}\cup\{wt\},\mathscr{V})$.

A substitution σ is said to *satisfy* an equation $\tau_1=_s\tau_2$ if $\sigma\tau_1=\sigma\tau_2$. Moreover, σ is said to *satisfy* a subtype relation $\tau_1<:_s\tau_2$ if $\sigma\tau_1<:_s\sigma\tau_2$.

Thus, σ is a *solution* for C if it satisfies all constraints in C. This is written $\sigma\models C$. The set $\mathscr{V}(C)$ denotes the set of type variables in C.

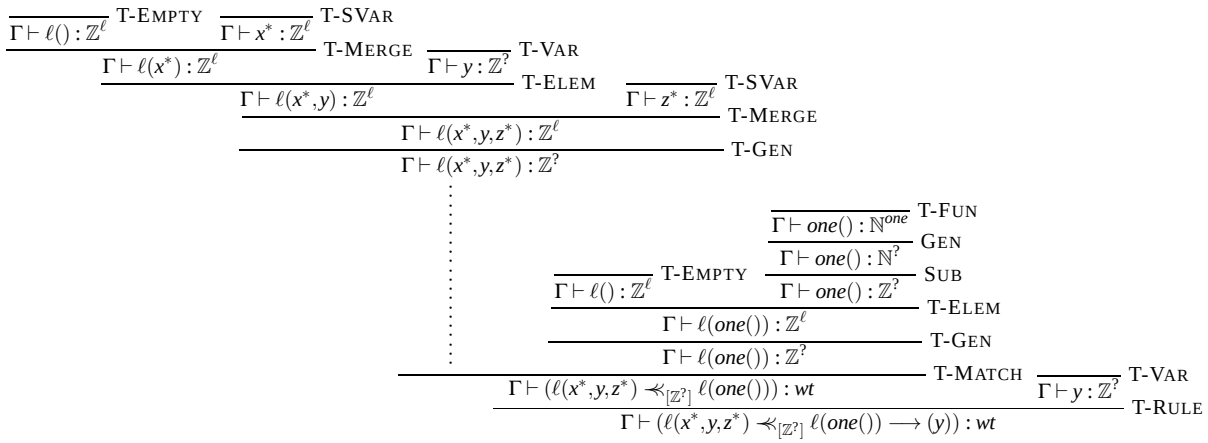

Figure 3: Type checking example.

Constraints are calculated according to the application of rules of type inference system (see Fig. 4) where we can read the judgment $\Gamma \vdash_{ct} e : \tau \bullet C$ as "the term e has type τ under assumptions Γ whenever the constraints C are satisfied". More formally, this judgment states that $\forall \sigma \,.\, (\sigma \models C \rightarrow \sigma\Gamma \vdash e : \sigma\tau)$.

3.1 Type inference algorithm

In Fig. 4 we give a type inference system with constraints. In order to infer the type of a given expression π, the context Γ is initialized to: 1) subtype declarations of the form $s_1^? <:_s s_2^?$ where $s_1^?$ and $s_2^? \in \mathscr{D}$; 2) a pair of the form $(f : s_1^?, \ldots, s_n^? \rightarrow s^f)$ for each syntactic operator f occurring in π where $s_i^?, s^f \in \mathscr{D}$ for $i \in [1,n]$; 3) a pair of the form $(\ell : s_1^{?*} \rightarrow s^\ell)$ for each variadic operator ℓ occurring in π where $s_1^?, s^\ell \in \mathscr{D}$; 4) a pair of the form $(x : \alpha)$ for each variable x occurring in π where $\alpha \in \mathscr{V}$ is a fresh type variable; 5) a pair of the form $(x^* : \alpha)$ for each star variable x^* occurring in π where $\alpha \in \mathscr{V}$ is a fresh type variable. Moreover, each type variable introduced in a sub-derivation is a fresh type variable and the fresh type variables in different sub-derivations are distinct. As in Section 2.2, we explain the rules concerning lists: [CT-EMPTY] infers for an empty list $\ell()$ a type variable α with the constraint $\alpha = s^\ell$, s^ℓ given by the rank of ℓ; [CT-ELEM] treats applications of lists to elements which are neither lists with the same function symbol nor star variables; [CT-MERGE] is applied to concatenate two lists of same type s^ℓ; and [CT-STAR] is applied to concatenate a list and a star variable of the same type s^ℓ.

Example 3.1. *Let* $\Gamma = \{\ell : (\mathbb{Z}^?)^* \rightarrow \mathbb{Z}^\ell, one : \rightarrow \mathbb{N}^{one}, x^* : \alpha_1, y : \alpha_2, z^* : \alpha_3, \mathbb{N}^? <:_s \mathbb{Z}^?\}$. *Then the expression* $\ell(x^*, y, z^*) \prec\!\!\prec_{[\alpha_4]} \ell(one()) \longrightarrow (y)$ *is well-typed and the deduction tree is given in Fig. 5.*

3.2 Constraint resolution

In Fig. 6 we propose an algorithm to decide whether a given constraint set C has a solution, where $g_1, g_2 \in \mathscr{F} \cup \mathscr{F}_v \cup \{?\}$. We denote by $s^{g_1} <:_s s'^{g_2} \in \Gamma$ the fact that there exists $s_1, \ldots, s_n$ such that $s^? <: s_1^? \in \Gamma$, $s_1^? <: s_2^? \in \Gamma, \ldots, s_n^? <: s'^? \in \Gamma$ and ($g_1 = g_2$ or $g_2 = ?$). If the algorithm stops without failure then C is said to be in *solved form*.

While solving a constraint set C we wish to make sure, after each application of a constraint resolution rule, that the constraint set at hand is satisfiable, so as to detect errors as soon as possible. Therefore we

$$\frac{}{\Gamma(x:\tau)\vdash_{ct} x:\alpha\bullet\{\alpha=_s\tau\}}\ \text{CT-VAR} \qquad \frac{}{\Gamma(x^*:\alpha_1)\vdash_{ct} x^*:\alpha\bullet\{\alpha_1=_s\alpha\}}\ \text{CT-SVAR}$$

$$\frac{\Gamma\vdash_{ct} e_1:\alpha_1\bullet C_1 \quad \ldots \quad \Gamma\vdash_{ct} e_n:\alpha_n\bullet C_n}{\Gamma(f:s_1^?,\ldots,s_n^?\to s^f)\vdash_{ct} f(e_1,\ldots,e_n):\alpha\bullet\{\alpha=_s s^f\}\bigcup_{i=1}^{n} C_i\cup\{\alpha_i<:_s s_i^?\}}\ \text{CT-FUN}$$

$$\frac{}{\Gamma(\ell:(s_1^?)^*\to s^\ell)\vdash_{ct}\ell():\alpha\bullet\{\alpha=_s s^\ell\}}\ \text{CT-EMPTY}$$

$$\frac{\Gamma\vdash_{ct}\ell(e_1,\ldots,e_n):\alpha\bullet C_1 \qquad \Gamma\vdash_{ct} e:\alpha_1\bullet C_2}{\Gamma(\ell:(s_1^?)^*\to s^\ell)\vdash_{ct}\ell(e_1,\ldots,e_n,e):\alpha\bullet\{\alpha=_s s^\ell,\alpha_1<:_s s_1^?\}\cup C_1\cup C_2}\ \text{CT-ELEM}$$
$$\text{if } \texttt{sortOf}(\Gamma,e)\neq s^\ell \text{ and } e\neq x^*$$

$$\frac{\Gamma\vdash_{ct}\ell(e_1,\ldots,e_n):\alpha\bullet C_1 \qquad \Gamma\vdash_{ct} e:\alpha\bullet C_2}{\Gamma(\ell:(s_1^?)^*\to s^\ell)\vdash_{ct}\ell(e_1,\ldots,e_n,e):\alpha\bullet\{\alpha=_s s^\ell\}\cup C_1\cup C_2}\ \text{CT-MERGE}$$
$$\text{if } \texttt{sortOf}(\Gamma,e)=s^\ell$$

$$\frac{\Gamma\vdash_{ct}\ell(e_1,\ldots,e_n):\alpha\bullet C_1 \qquad \Gamma\vdash_{ct} x^*:\alpha\bullet C_2}{\Gamma(\ell:(s_1^?)^*\to s^\ell)\vdash_{ct}\ell(e_1,\ldots,e_n,x^*):\alpha\bullet\{\alpha=_s s^\ell\}\cup C_1\cup C_2}\ \text{CT-STAR}$$

$$\frac{\Gamma\vdash_{ct} e_1:\alpha_1\bullet C_1 \qquad \Gamma\vdash_{ct} e_2:\alpha_2\bullet C_2}{\Gamma\vdash_{ct}(e_1\ \prec\!\!\!\prec_{[\tau]}\ e_2):wt\bullet\{\alpha_1<:_s\tau,\alpha_2=_s\tau\}\cup C_1\cup C_2}\ \text{CT-MATCH}$$

$$\frac{\Gamma\vdash_{ct}(cond_1):wt\bullet C_1 \qquad \ldots \qquad \Gamma\vdash_{ct}(cond_n):wt\bullet C_n}{\Gamma\vdash_{ct}(cond_1\wedge\ldots\wedge cond_n):wt\bullet\bigcup_{i=1}^{n} C_i}\ \text{CT-CONJ}$$

$$\frac{\Gamma\vdash_{ct}(cond):wt\bullet C_{cond} \qquad \Gamma\vdash_{ct} e_1:\tau_1\bullet C_1 \quad \ldots \quad \Gamma\vdash_{ct} e_n:\tau_n\bullet C_n}{\Gamma\vdash_{ct}(cond\longrightarrow(e_1,\ldots,e_n)):wt\bullet C_{cond}\bigcup_{i=1}^{n} C_i}\ \text{CT-RULE}$$
$$\text{if } \texttt{sortOf}(\Gamma,e_i)=\tau_i, \text{ for } i\in[1,n] \text{ where } \tau_i\in\mathscr{T}_{ype}(\mathscr{D}\cup\{wt\},\mathscr{V})$$

Figure 4: Type inference rules.

must combine the rules for error detection and constraint resolution in order to keep C in solved form. The rules for the constraint resolution algorithm are provided in Fig. 7, where $g,g_1,g_2\in\mathscr{F}\cup\mathscr{F}_v\cup\{?\}$. The rules (1)-(14) are recursively applied over C. More precisely, rules (1)-(3) work as a garbage collector removing constraints that are no more useful. Rules (4) and (5) generate σ. Rules (6) and (7) generate more simplified constraints. Rules (8)-(12) generate σ and simplified constraints by antisymmetric and transitive subtype closure. Rules (13) and (14) are applied when none of previous rules can be applied generating a new σ from a constraint over a type variable that has no other constraints. The algorithm

$$\dfrac{\dfrac{}{\Gamma\vdash_{ct}\ell():\alpha_5\bullet C_3=\{\alpha_5=_s\mathbb{Z}^\ell\}}\text{CT-EMPTY}\quad\dfrac{}{\Gamma\vdash_{ct}x^*:\alpha_5\bullet C_4=\{\alpha_5=_s\alpha_1\}}\text{CT-SVAR}}{\Gamma\vdash_{ct}\ell(x^*):\alpha_5\bullet C_2=\{\alpha_5=_s\mathbb{Z}^\ell\}\cup C_3\cup C_4}\text{CT-STAR}\quad\dfrac{}{\Gamma\vdash_{ct}y:\alpha_8\bullet C_5=\{\alpha_8=_s\alpha_2\}}\text{CT-VAR}$$
$$\dfrac{\Gamma\vdash_{ct}\ell(x^*):\alpha_5\bullet C_2=\{\alpha_5=_s\mathbb{Z}^\ell\}\cup C_3\cup C_4\qquad\Gamma\vdash_{ct}y:\alpha_8\bullet C_5=\{\alpha_8=_s\alpha_2\}}{\Gamma\vdash_{ct}\ell(x^*,y):\alpha_5\bullet C_1=\{\alpha_5=_s\mathbb{Z}^\ell,\alpha_8<:_s\mathbb{Z}^?\}\cup C_2\cup C_5}\text{CT-ELEM}$$
$$\vdots$$
$$\dfrac{\Gamma\vdash_{ct}\ell(x^*,y):\alpha_5\bullet C_1=\{\alpha_5=_s\mathbb{Z}^\ell,\alpha_8<:_s\mathbb{Z}^?\}\cup C_2\cup C_5\qquad\dfrac{}{\Gamma\vdash_{ct}z^*:\alpha_5\bullet C_6=\{\alpha_5=_s\alpha_3\}}\text{CT-SVAR}}{\Gamma\vdash_{ct}\ell(x^*,y,z^*):\alpha_5\bullet C_p=\{\alpha_5=_s\mathbb{Z}^\ell\}\cup C_1\cup C_6}\text{CT-STAR}$$

(1)

$$\dfrac{\dfrac{}{\Gamma\vdash_{ct}\ell():\alpha_6\bullet C_7=\{\alpha_6=_s\mathbb{Z}^\ell\}}\text{CT-EMPTY}\quad\dfrac{}{\Gamma\vdash_{ct}one():\alpha_7\bullet C_8\{\alpha_7=_s\mathbb{N}^{one}\}}\text{CT-FUN}}{\Gamma\vdash_{ct}\ell(one()):\alpha_6\bullet C_s=\{\alpha_6=_s\mathbb{Z}^\ell,\alpha_7<:_s\mathbb{Z}^?\}\cup C_7\cup C_8}\text{CT-ELEM}$$

(2)

$$\dfrac{\overset{\textbf{(1)}}{\Gamma\vdash_{ct}(\ell(x^*,y,z^*)\prec_{[\alpha_4]}\ell(one())):wt\bullet C_{cond}=\{\alpha_5<:_s\alpha_4,\alpha_6=_s\alpha_4\}}\text{CT-MATCH}\quad\overset{\textbf{(2)}}{\dfrac{}{\Gamma\vdash_{ct}y:\alpha_9\bullet C_{10}=\{\alpha_9=_s\alpha_2\}}\text{CT-VAR}}}{\Gamma\vdash_{ct}(\ell(x^*,y,z^*)\prec_{[\alpha_4]}\ell(one())\longrightarrow(y)):wt\bullet C_r=\{\alpha_2=_s\alpha_2\}\cup C_{cond}\cup C_{10}}\text{CT-RULE}$$

Figure 5: Type inference example.

$$\begin{array}{lll}
(1) & \{s_1^{g_1}<:_s\alpha,\alpha<:_s s_2^{g_2}\}\uplus C' & \Longrightarrow \quad fail \text{ if } s_1^{g_1}<:_s s_2^{g_2}\notin\Gamma\\
(2) & \{s_1^{g_1}<:_s\alpha,s_2^{g_2}<:_s\alpha\}\uplus C' & \Longrightarrow \quad fail \text{ if } \nexists s.(s_1^{g_1}<:_s s^?\in\Gamma\wedge s_2^{g_2}<:_s s^?\in\Gamma)\\
(3) & \{\alpha<:_s s_1^{g_1},\alpha<:_s s_2^{g_2}\}\uplus C' & \Longrightarrow \quad fail \text{ if } (s_1^{g_1}<:_s s_2^{g_2}\notin\Gamma\wedge s_2^{g_2}<:_s s_1^{g_1}\notin\Gamma)\\
(4) & \{s_1^{g_1}<:_s s_2^{g_2}\}\uplus C' & \Longrightarrow \quad fail \text{ if } s_1^{g_1}<:_s s_2^{g_2}\notin\Gamma\\
(5) & \{s_1^{g_1}=s_2^{g_2}\}\uplus C' & \Longrightarrow \quad fail \text{ if } s_1\neq s_2\vee g_1\neq g_2
\end{array}$$

Figure 6: Rules for detection of errors in a constraint set C.

stops if: a rule returns $C=\varnothing$, then the algorithm returns the solution σ; if C reaches a non-solved form, then the algorithm for detection of errors returns $fail$; or if C reaches a normal form different from the empty set, then the algorithm returns an error. We say that the algorithm is *failing* if it returns either *fails* or an error.

Example 3.2. *Let* $\Gamma=\{\ell:(\mathbb{Z}^?)^*\to\mathbb{Z}^\ell,one:\to\mathbb{N}^{one},x^*:\alpha_1,y:\alpha_2,z^*:\alpha_3,\mathbb{N}^?<:_s\mathbb{Z}^?\}$ *and* $C_{cond}=\{\alpha_5=_s\mathbb{Z}^\ell,\alpha_{10}=_s\alpha_1,\alpha_5=_s\mathbb{Z}^\ell,\alpha_{10}=_s\mathbb{Z}^\ell,\alpha_9=_s\alpha_2,\alpha_5=_s\mathbb{Z}^\ell,\alpha_9<:_s\mathbb{Z}^?,\alpha_8=_s\alpha_3,\alpha_5=_s\mathbb{Z}^\ell,\alpha_8=_s\mathbb{Z}^\ell,\alpha_6=_s\mathbb{Z}^\ell,\alpha_7=_s\mathbb{N}^{one},\alpha_6=_s\mathbb{Z}^\ell,\alpha_7<:_s\mathbb{Z}^?,\alpha_5<:_s\alpha_4,\alpha_6=_s\alpha_4,\alpha_2=_s\alpha_2\}$ *from the Example 3.1. Let* $\sigma=\varnothing$ *and* $C=C_{cond}$. *The constraint resolution algorithm starts by:*

1. *Application of sequence of rules (4), (1) and (5) generating* $\{\alpha_2<:_s\mathbb{Z}^?,\mathbb{N}^{one}<:_s\mathbb{Z}^?,\mathbb{Z}^\ell<:_s\mathbb{Z}^\ell\}\cup C$ *and* $\{\alpha_5\mapsto\mathbb{Z}^\ell,\alpha_{10}\mapsto\alpha_1,\alpha_1\mapsto\mathbb{Z}^\ell,\alpha_9\mapsto\alpha_2,\alpha_8\mapsto\alpha_3,\alpha_3\mapsto\mathbb{Z}^\ell,\alpha_6\mapsto\mathbb{Z}^\ell,\alpha_7\mapsto\mathbb{N}^{one},\alpha_4\mapsto\mathbb{Z}^\ell\}\cup\sigma$
2. *Application of rules (1), (2) and (3) generating* $\{\alpha_2<:_s\mathbb{Z}^?\}$ *and* σ*;*

$$
\begin{array}{lll}
(1) & \{\tau =_s \tau\} \uplus C', \sigma & \Longrightarrow \quad C', \sigma \\
(2) & \{\tau <:_s \tau\} \uplus C', \sigma & \Longrightarrow \quad C', \sigma \\
(3) & \{s_1^{g_1} <:_s s_2^{g_2}\} \uplus C', \sigma & \Longrightarrow \quad C', \sigma \text{ if } s_1^{g_1} <:_s s_2^{g_2} \in \Gamma \\
 & & \\
(4) & \{\alpha =_s \tau\} \uplus C', \sigma & \Longrightarrow \quad [\alpha \mapsto \tau]C', \{\alpha \mapsto \tau\} \cup \sigma \\
(5) & \{\tau =_s \alpha\} \uplus C', \sigma & \Longrightarrow \quad [\alpha \mapsto \tau]C', \{\alpha \mapsto \tau\} \cup \sigma \\
 & & \\
(6) & \{s_1^{g_1} <:_s \alpha, s_2^{g_2} <:_s \alpha\} \uplus C', \sigma & \Longrightarrow \quad \{s^? <:_s \alpha\} \cup C', \sigma \text{ if } \exists s \centerdot (s_1^{g_1} <:_s s^? \in \Gamma \wedge s_2^{g_2} <:_s s^? \in \Gamma) \\
(7a) & \{\alpha <:_s s_1^{g_1}, \alpha <:_s s_2^{g_2}\} \uplus C', \sigma & \Longrightarrow \quad \{\alpha <:_s s_1^{g_1}\} \cup C', \sigma \text{ if } (s_1^{g_1} <:_s s_2^{g_2} \in \Gamma) \\
(7b) & \{\alpha <:_s s_1^{g_1}, \alpha <:_s s_2^{g_2}\} \uplus C', \sigma & \Longrightarrow \quad \{\alpha <:_s s_2^{g_2}\} \cup C', \sigma \text{ if } (s_2^{g_2} <:_s s_1^{g_1} \in \Gamma) \\
 & & \\
(8) & \{\tau_1 <:_s \tau_2, \tau_2 <:_s \tau_1\} \uplus C', \sigma & \Longrightarrow \quad \{\tau_1 =_s \tau_2\} \cup C', \sigma \\
(9) & \{\alpha_1 <:_s \alpha, \alpha <:_s \alpha_2\} \uplus C', \sigma & \Longrightarrow \quad \{\alpha_1 <:_s \alpha_2\} \cup [\alpha \mapsto \alpha_2]C', \{\alpha \mapsto \alpha_2\} \cup \sigma \\
(10) & \{s^g <:_s \alpha, \alpha <:_s \alpha_1\} \uplus C', \sigma & \Longrightarrow \quad \{s^g <:_s \alpha_1\} \cup [\alpha \mapsto \alpha_1]C', \{\alpha \mapsto \alpha_1\} \cup \sigma \\
(11) & \{\alpha_1 <:_s \alpha, \alpha <:_s s^g\} \uplus C', \sigma & \Longrightarrow \quad \{\alpha_1 <:_s s^g\} \cup [\alpha \mapsto \alpha_1]C', \{\alpha \mapsto \alpha_1\} \cup \sigma \\
(12) & \{s_1^{g_1} <:_s \alpha, \alpha <:_s s_2^{g_2}\} \uplus C', \sigma & \Longrightarrow \quad [\alpha \mapsto s_2^{g_2}]C', \{\alpha \mapsto s_2^{g_2}\} \cup \sigma \text{ if } s_1^{g_1} <:_s s_2^{g_2} \in \Gamma \\
 & & \\
(13) & \{\alpha <:_s \tau\} \uplus C', \sigma & \Longrightarrow \quad C', \{\alpha \mapsto \tau\} \cup \sigma \text{ if } \alpha \notin \mathscr{V}(C') \\
(14) & \{\tau <:_s \alpha\} \uplus C', \sigma & \Longrightarrow \quad C', \{\alpha \mapsto \tau\} \cup \sigma \text{ if } \alpha \notin \mathscr{V}(C')
\end{array}
$$

Figure 7: Constraint resolution rules in context Γ.

3. *Application of rule (13) generating $\varnothing$ and $\{\alpha_2 \mapsto \mathbb{Z}^?\} \cup \sigma$, the algorithm then stops and returns σ providing a substitution for all type variables in the deduction tree of $\ell(x^*, y, z^*) \prec\!\!\prec_{[\alpha_4]} \ell(one()) \longrightarrow (y)$.*

4 Properties

Since our type checking system and our type inference system address the same issue, we must check two properties. First, we show that every typing judgment that can be derived from the inference rules also follows from the checking rules (Theorem 4.2), in particular the soundness. Then we show that a solution given by the checking rules can be extended to a solution proposed by the inference rules (Theorem 4.4).

Definition 4.1 (Solution)**.** *Let Γ be a context and e a term.*

- *A* solution *for (Γ, e) is a pair (σ, T_1) such that $\sigma\Gamma \vdash \sigma e : T_1$, where $T_1 \in \mathscr{D} \cup \{wt\}$.*
- *Assuming a well-formed sequent $\Gamma \vdash e : \tau \bullet C$, a* solution *for (Γ, e, τ, C) is a pair (σ, T_2) such that σ satisfies C and $\sigma\tau <:_s T_2$, where $T_2 \in \mathscr{D} \cup \{wt\}$ and $\tau \in \mathscr{T}_{ype}(\mathscr{D} \cup \{wt\}, \mathscr{V})$.*

Theorem 4.2 (Soundness of constraint typing)**.** *Suppose that $\Gamma \vdash_{ct} e : \tau \bullet C$ is a valid sequent. If (σ, s^g) is a solution for (Γ, e, τ, C), then it is also a solution for (Γ, e) (i.e. e is well-typed in Γ).*

Proof. By induction on the given constraint typing derivation for $\Gamma \vdash_{ct} e : \tau \bullet C$. We just detail the most noteworthy cases of this proof.

Case CT-ELEM: $e = \ell(a_1,\ldots,a_n,a)$ $\quad \tau = \alpha$
$\Gamma \vdash_{ct} \ell(a_1,\ldots,a_n) : \alpha \bullet C_1$ $\quad \Gamma \vdash_{ct} a : \alpha_1 \bullet C_2$
$C = C_1 \cup C_2 \cup \{\alpha =_s s_2^\ell, \alpha_1 <:_s s_1^?\}$

We are given that (σ, s^g) is a solution for $(\Gamma(\ell : (s_1^?)^* \to s_2^\ell), e, \alpha, C)$, that is, σ satisfies C and $\sigma\alpha <:_s s^g$. Since (σ, s^g) satisfies C_1 and C_2, $(\sigma, \sigma\alpha)$ and $(\sigma, \sigma\alpha_1)$ are solutions for $(\Gamma, \ell(a_1,\ldots,a_n), \alpha, C_1)$ and $(\Gamma, a, \alpha_1, C_2)$, respectively. By the induction hypothesis, we have $\sigma\Gamma \vdash \sigma(\ell(a_1,\ldots,a_n)) : \sigma\alpha$ and $\sigma\Gamma \vdash \sigma a : \sigma\alpha_1$. Since $\sigma\alpha_1 <:_s s_1^?$, by SUB we obtain $\sigma\Gamma \vdash \sigma a : s_1^?$. Since $\sigma\alpha = s_2^\ell$, by T-ELEM we obtain $\sigma(\Gamma(\ell : (s_1^?)^* \to s_2^\ell)) \vdash \sigma(\ell(a_1,\ldots,a_n,a)) : s_2^\ell$. By SUB we obtain $\sigma(\Gamma(\ell : (s_1^?)^* \to s_2^\ell)) \vdash \sigma(\ell(a_1,\ldots,a_n,a)) : s^g$, as required.

Case CT-MERGE: $e = \ell(a_1,\ldots,a_n,a)$ $\quad \tau = \alpha$
$\Gamma \vdash_{ct} \ell(a_1,\ldots,a_n) : \alpha \bullet C_1$ $\quad \Gamma \vdash_{ct} a_1 : \alpha \bullet C_2$
$C = C_1 \cup C_2 \cup \{\alpha =_s s_2^\ell\}$

We are given that (σ, s^g) is a solution for $(\Gamma(\ell : (s_1^?)^* \to s_2^\ell), e, \alpha, C)$, that is, σ satisfies C and $\sigma\alpha <:_s s^g$. Since (σ, s^g) satisfies C_1 and C_2, $(\sigma, \sigma\alpha)$ and $(\sigma, \sigma\alpha_1)$ are solutions for $(\Gamma, \ell(a_1,\ldots,a_n), \alpha, C_1)$ and (Γ, a, α, C_2). By the induction hypothesis, we have $\sigma\Gamma \vdash \sigma(\ell(a_1,\ldots,a_n)) : \sigma\alpha$ and $\sigma\Gamma \vdash \sigma a : \sigma\alpha_1$. Since $\sigma\alpha = s_2^\ell$, by T-MERGE we obtain $\sigma(\Gamma(\ell : (s_1^?)^* \to s_2^\ell)) \vdash \sigma(\ell(a_1,\ldots,a_n,a)) : s_2^\ell$. By SUB we obtain $\sigma(\Gamma(\ell : (s_1^?)^* \to s_2^\ell)) \vdash \sigma(\ell(a_1,\ldots,a_n,a)) : s^g$, as required.

Case CT-MATCH: $e = a_1 \ \prec_{[\tau_1]} \ a_2$ $\quad \tau = wt$
$\Gamma \vdash_{ct} a_1 : \alpha_1 \bullet C_1$ $\quad \Gamma \vdash_{ct} a_2 : \alpha_2 \bullet C_2$
$C = C_1 \cup C_2 \cup \{\alpha_1 <:_s \tau_1, \alpha_2 =_s \tau_1\}$

We are given that (σ, wt) is a solution for (Γ, e, wt, C), that is, σ satisfies C and $\sigma wt <:_s wt$. Since (σ, wt) satisfies C_1 and C_2, $(\sigma, \sigma\alpha_1)$ and $(\sigma, \sigma\alpha_2)$ are solutions for $(\Gamma, a_1, \alpha_1, C_1)$ and $(\Gamma, a_2, \alpha_2, C_2)$, respectively. By the induction hypothesis, we have $\sigma\Gamma \vdash \sigma a_1 : \sigma\alpha_1$ and $\sigma\Gamma \vdash \sigma a_2 : \sigma\alpha_2$. Since $\sigma\alpha_1 <:_s \sigma\tau_1$, by SUB we obtain $\sigma\Gamma \vdash \sigma a_1 : \sigma\tau_1$. Since $\sigma\alpha_2 = \sigma\tau_1$, by T-MATCH we obtain $\sigma\Gamma \vdash \sigma(a_1 \ \prec_{[\tau_1]} \ a_2) : wt$, as required. □

Definition 4.3 (Normal form of typing derivation)**.** *A typing derivation is in* normal form *if it does not have successive applications of rule* [SUB].

Theorem 4.4 (Completeness of constraint typing)**.** *Suppose that* $\pi = \Gamma \vdash_{ct} e : \tau \bullet C$*. Write* $V(\pi)$ *for the set of all type variables mentioned in the last rule used to derive* π *and write* $\sigma\backslash V(\pi)$ *for the substitution that is undefined for all the variables in* $V(\pi)$ *and otherwise behaves like* σ*. If* (σ, s^g) *is a solution for* (Γ, e) *and* $dom(\sigma) \cap V(\pi) = \varnothing$*, then there is some solution* (σ', s^g) *for* (Γ, e, τ, C) *such that* $\sigma'\backslash V(\pi) = \sigma$*.*

Proof. By induction on the given constraint typing derivation in normal form, but we must take care with fresh names of variables. We just detail the most noteworthy cases of this proof.

Case CT-ELEM: $e = \ell(a_1,\ldots,a_n,a)$ $\quad \tau = \alpha$
$\pi_1 = \Gamma \vdash_{ct} \ell(a_1,\ldots,a_n) : \alpha \bullet C_1$ $\quad \pi_2 = \Gamma \vdash_{ct} a : \alpha_1 \bullet C_2$
$C = C_1 \cup C_2 \cup \{\alpha =_s s_2^\ell, \alpha_1 <:_s s_1^?\}$ $\quad V(\pi) = \{\alpha, \alpha_1\}$
$\texttt{sortOf}(\Gamma, a) \neq s_2^\ell$

From the assumption that (σ, s^g) is a solution for $(\Gamma(\ell : (s_1^?)^* \to s_2^\ell), \ell(a_1,\ldots,a_n,a))$ and $dom(\sigma) \cap V(\pi) = \varnothing$, we have $\sigma(\Gamma(\ell : (s_1^?)^* \to s_2^\ell)) \vdash \sigma(\ell(a_1,\ldots,a_n,a)) : s^g$. This can be derived from: 1) T-MERGE, 2) T-ELEM or 3) SUB. In all those cases, we must exhibit a substitution σ' such that: (a) $\sigma' \backslash V(\pi)$ agrees with σ; (b) $\sigma'\alpha <:_s s^g$; (c) σ' satisfies C_1 and C_2; and (d) σ' satisfies $\{\alpha =_s s_2^\ell, \alpha_1 <:_s s_1^?\}$. We reason by cases as follows:

1. By T-MERGE we assume that $s^g = s_2^\ell$ and we know that $\sigma\Gamma \vdash \sigma(\ell(a_1,\ldots,a_n)) : s_2^\ell$ and $\sigma\Gamma \vdash \sigma a : s_2^\ell$. But since we cannot find a type s_3^ℓ such that $s_3^\ell <:_s s_2^\ell$, $\sigma\Gamma \vdash \sigma a : s_2^\ell$ cannot be derived even from SUB. Thus T-MERGE is not a relevant case.
2. By T-ELEM we assume that $s^g = s_2^\ell$ and we know that $\sigma\Gamma \vdash \sigma(\ell(a_1,\ldots,a_n)) : s_2^\ell$ and $\sigma\Gamma \vdash \sigma a : s_1^?$. By the induction hypothesis, there are solutions (σ_1, s_2^ℓ) for $(\Gamma, \ell(a_1,\ldots,a_n), \alpha, C_1)$ and $(\sigma_2, s_1^?)$ for $(\Gamma, a, \alpha_1, C_2)$, and $dom(\sigma_1)\backslash V(\pi_1) = \varnothing = dom(\sigma_2)\backslash V(\pi_2)$. Define $\sigma' = \{\alpha \mapsto s_2^\ell, \alpha_1 \mapsto s_1^?\} \cup \sigma \cup \sigma_1 \cup \sigma_2$. Conditions (a), (b), (c) and (d) are obviously satisfied. Thus, we see that (σ', s^g) is a solution for $(\Gamma(\ell : (s_1^?)^* \to s_2^\ell), \ell(a_1,\ldots,a_n,a), \alpha, C)$.
3. By SUB we assume that $s_2^\ell <:_s s^g \in \Gamma$ and we know that $\sigma(\Gamma(\ell : (s_1^?)^* \to s_2^\ell)) \vdash \sigma(\ell(a_1,\ldots,a_n,a)) : s_2^\ell$. This must be derived from T-ELEM, similar to case (2).

Case CT-MERGE:
$e = \ell(a_1,\ldots,a_n,a)$ $\quad$ $\tau = \alpha$
$\pi_1 = \Gamma \vdash_{ct} \ell(a_1,\ldots,a_n) : \alpha \bullet C_1$ $\quad$ $\pi_2 = \Gamma \vdash_{ct} a : \alpha \bullet C_2$
$C = C_1 \cup C_2 \cup \{\alpha =_s s_2^\ell\}$ $\quad$ $V(\pi) = \{\alpha\}$
$\mathtt{sortOf}(\Gamma, a) = s_2^\ell$

From the assumption that (σ, s^g) is a solution for $(\Gamma(\ell : (s_1^?)^* \to s_2^\ell), \ell(a_1,\ldots,a_n,a))$ and $dom(\sigma) \cap V(\pi) = \varnothing$, we have $\sigma(\Gamma(\ell : (s_1^?)^* \to s_2^\ell)) \vdash \sigma(\ell(a_1,\ldots,a_n,a)) : s^g$. This can be derived from: 1) T-MERGE, 2) T-ELEM or 3) SUB. In all those cases, we must exhibit a substitution σ' such that: (a) $\sigma' \backslash V(\pi)$ agrees with σ; (b) $\sigma'\alpha <:_s s^g$; (c) σ' satisfies C_1 and C_2; and (d) σ' satisfies $\{\alpha =_s s_2^\ell\}$. We reason by cases as follows:

1. By T-MERGE we assume that $s^g = s_2^\ell$ and we know that $\sigma\Gamma \vdash \sigma(\ell(a_1,\ldots,a_n)) : s_2^\ell$ and $\sigma\Gamma \vdash \sigma a : s_2^\ell$. By the induction hypothesis, there are solutions (σ_1, s_2^ℓ) for $(\Gamma, \ell(a_1,\ldots,a_n), \alpha, C_1)$ and (σ_2, s_2^ℓ) for (Γ, a, α, C_2), and $dom(\sigma_1)\backslash V(\pi_1) = \varnothing = dom(\sigma_2)\backslash V(\pi_2)$. Define $\sigma' = \{\alpha \mapsto s_2^\ell\} \cup \sigma \cup \sigma_1 \cup \sigma_2$. Conditions (a), (b), (c) and (d) are obviously satisfied. Thus, we see that (σ', s^g) is a solution for $(\Gamma(\ell : (s_1^?)^* \to s_2^\ell), \ell(a_1,\ldots,a_n,a), \alpha, C)$.
2. By T-ELEM we assume that $s^g = s_2^\ell$ and we know that $\sigma\Gamma \vdash \sigma(\ell(a_1,\ldots,a_n)) : s_2^\ell$ and $\sigma\Gamma \vdash \sigma a : s_1^?$. But, because of the application condition of T-ELEM, we cannot find a type s_1^ℓ for σa such that $s_1^\ell <:_s s_1^?$, $\sigma\Gamma \vdash \sigma a : s_1^?$ cannot be derived from GEN. Likewise, since we cannot find a type s_3^ℓ for σa such that $s_3^\ell <:_s s_1^?$, $\sigma\Gamma \vdash \sigma a : s_1^?$ cannot be derived even from SUB. Thus T-ELEM is not a relevant case.
3. By SUB we assume that $s_2^\ell <:_s s^g \in \Gamma$ and we know that $\sigma(\Gamma(\ell : (s_1^?)^* \to s_2^\ell)) \vdash \sigma(\ell(a_1,\ldots,a_n,a)) : s_2^\ell$. This must be derived from T-MERGE, similar to case (1).

Case CT-MATCH:
$e = a_1 \ \prec_{[\tau_1]} \ a_2$ $\quad$ $\tau = wt$
$\pi_1 = \Gamma \vdash_{ct} a_1 : \alpha_1 \bullet C_1$ $\quad$ $\pi_2 = \Gamma \vdash_{ct} a_2 : \alpha_2 \bullet C_2$
$C = C_1 \cup C_2 \cup \{\alpha_1 <:_s \tau_1, \alpha_2 =_s \tau_1\}$ $\quad$ $V(\pi) = \{\alpha_1, \alpha_2, \tau_1\}$ if $\tau_1 \in \mathscr{V}$
$V(\pi) = \{\alpha_1, \alpha_2\}$ if $\tau_1 \notin \mathscr{V}$

From the assumption that (σ, wt) is a solution for $(\Gamma, a_1 \ll_{[\tau_1]} a_2)$ and $dom(\sigma) \cap V(\pi) = \varnothing$, we have $\sigma\Gamma \vdash \sigma(a_1 \ll_{[\tau_1]} a_2) : wt$. This must be derived from T-MATCH, we know that $\sigma\Gamma \vdash \sigma a_1 : \sigma\tau_1$ and $\sigma\Gamma \vdash \sigma a_2 : \sigma\tau_1$. By the induction hypothesis, there are solutions $(\sigma_1, \sigma\tau_1)$ for $(\Gamma, a_1, \alpha_1, C_1)$ and $(\sigma_2, \sigma\tau_1)$ for $(\Gamma, a_2, \alpha_2, C_2)$. We must exhibit a substitution σ' such that: (a) $\sigma' \backslash V(\pi)$ agrees with σ; (b) $\sigma' wt <:_s wt$; (c) σ' satisfies C_1 and C_2; and (d) σ' satisfies $\{\alpha_1 <:_s \tau_1, \alpha_2 =_s \tau_1\}$. Define $\sigma'' = \{\alpha_1 \mapsto s^g, \alpha_2 \mapsto s^g\} \cup \sigma \cup \sigma_1 \cup \sigma_2$, where $s^g \in \mathscr{D}$. Moreover, define $\sigma' = \sigma'' \cup \{\tau_1 \mapsto s^g\}$ if $\tau_1 \in \mathscr{V}$ and $\sigma' = \sigma''$ otherwise. Conditions (a), (b), (c) and (d) are obviously satisfied. Thus, we see that (σ', wt) is a solution for $(\Gamma, (a_1 \ll_{[\tau_1]} a_2), wt, C)$. □

The constraint resolution algorithm always terminates. More formally:

Theorem 4.5 (Termination of algorithm)**.**

1. *the algorithm halts, either by failing or by returning a substitution, for all C;*
2. *if the algorithm returns σ, then σ is a solution for C;*

We can already sketch a proof of Theorem 4.5 following Pierce [15].

Proof. For part 1, define the *degree* of a constraint set C to be the pair (m,n), where m is the number of constraints in C and n is the number of subtyping constraints in C. The algorithm terminates immediately (with success in the case of an empty constraint set or failure for an equation involving two different decorated sorts) or makes recursive calls to itself with a constraint set of lexicographically smaller degree.

For part 2, by induction on the number of recursive calls in the computation of the algorithm. □

5 Conclusion

In this paper we have presented a type system for the pattern matching constructs of Tom. The system is composed of type checking and type inference algorithms with subtyping over sorts. Since Tom also implements associative pattern matching over variadic operators, we were interested in defining both a way to distinguish these from syntactic operators and checking and inferring their types.

We have obtained the following: our type inference system is sound and complete w.r.t. checking, showed by Theorems 4.4 and 4.2. This is the first step towards an effective implementation, thus leading to a safer Tom. However, we still need to investigate type unicity that we believe to hold under our assumptions of non-overloading and non-multiple inheritance.

As we have considered a subset of the Tom language, future work will focus on extending the type system to handle the other constructions of the language such as anti-patterns [12, 13]. As a slightly more prospective research area, we also want parametric polymorphism over types for Tom: our type system will therefore have to be able to handle that as well.

Acknowledgements

We would like to acknowledge the numerous discussions we had in the Protheo and Pareo teams, especially with Paul Brauner, on these topics during these last years as well as the constructive and useful comments done by the anonymous refereees.

References

[1] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints (extended abstract). In *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340. IEEE Computer Society Press, 1992.

[2] H. Ait-Kaci, A. Podelski, and G. Smolka. A feature constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1-2):263–283, 1994.

[3] Emilie Balland. *Conception d'un langage dédié à l'analyse et la transformation de programmes*. PhD thesis, Université Henri Poincaré, 2009.

[4] Emilie Balland, Claude Kirchner, and Pierre-Etienne Moreau. Formal Islands. In Michael Johnson and Varmo Vene, editors, *11th International Conference on Algebraic Methodology and Software Technology*, volume 4019 of *LNCS*, pages 51–65. Springer, 2006.

[5] Gilles Barthe, Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Pure Patterns Type Systems. In *Principles of Programming Languages - POPL2003, New Orleans, USA*. ACM, January 2003.

[6] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[7] Hubert Comon. Completion of rewrite systems with membership constraints. part i: Deduction rules. *Journal of Symbolic Computation*, 25(4):397–419, 1998.

[8] Duggan Dominic. Finite subtype inference with explicit polymorphism. *Sci. Comput. Program.*, 39(1):57–92, 2001.

[9] K. Futatsugi, J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ-2. In B. Reid, editor, *Proceedings 12th ACM Symp. on Principles of Programming Languages*, pages 52–66. ACM, 1985.

[10] J. A. Goguen, Claude Kirchner, Hélène Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ-3. In J.-P. Jouannaud and S. Kaplan, editors, *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, July 1987.

[11] Claus Hintermeier, Claude Kirchner, and Hélène Kirchner. Dynamically-typed computations for order-sorted equational presentations. *Journal of Symbolic Computation*, 25(4):455–526, 19[7]98.

[12] Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-pattern matching. In *16th European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 110–124, Braga, Portugal, 2007. Springer.

[13] Radu Kopetz. *Contraintes d'anti-filtrage et programmation par réécriture*. PhD thesis, Institut National Polytechnique de Lorraine, 2008.

[14] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[15] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. Chapter 22.

[16] François Pottier. Simplifying subtyping constraints: a theory. *Inf. Comput.*, 170(2):153–183, 2001.

[17] Steffen van Bakel and Maribel Fernández. Normalization results for typeable rewrite systems. *Information and Computation*, 133(2):73–116, 1997.