A Type System for Tom

Claude Kirchner, Pierre-Etienne Moreau, Cláudia Tavares

A Type System for Tom

Claude Kirchner
INRIA Bordeaux - Sud Ouest
351, cours de la Libération, BP A29,
33405 Talence Cedex France
Claude.Kirchner@inria.fr

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

Cláudia Tavares*
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. But 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 addiction, 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* [3, 2] where the *ocean* consists of Java code and the *island* of algebraic patterns. For simplicity, we consider 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` 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 CAPES, BEX 4878-06-0, Brazil.

```
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 propose an algorithm based on unification of equality constraints [7] and simplification of subtype constraints [4, 1, 9]. 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.

2 Type checking

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

$$\begin{array}{lcl} \mathit{rule} & ::= & \mathit{cond} \longrightarrow \mathit{action} \\ \mathit{cond} & ::= & \mathit{term}_1 \prec\!\!\!\prec_{[s]} \mathit{term}_2 \mid \mathit{cond}_1 \wedge \mathit{cond}_2 \\ \mathit{term} & ::= & x \mid f(\mathit{term}_1, \ldots, \mathit{term}_n) \\ \mathit{action} & ::= & (x_1, \ldots, x_n) \end{array}$$

The left-hand side of a rule is a conjunction of matching conditions $\mathit{term}_1 \prec\!\!\!\prec_{[s]} \mathit{term}_2$ consisting of a pair of terms and where s denotes a sort. Since we allow for some symbols to be associative, we introduce two kinds of symbols. Fixed arity ones to denote free symbols and variadic symbols to denote associative ones. We denote these two kinds of symbol sets $\mathscr{F}$ and $\mathscr{F}_v$ respectively. Terms are many-sorted variadic terms composed of variables $x \in \mathscr{X}$ and function symbols $f \in \mathscr{F} \cup \mathscr{F}_v$. The set of terms is written $\mathscr{T}(\mathscr{F} \cup \mathscr{F}_v, \mathscr{X})$. In the following, we often write l a variadic operator and call it a *list*. In general, an *action* is a Java statement, but for our purpose we can consider an abstraction described by the variables $x_1, \ldots, x_n \in \mathscr{X}$ 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:*

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

In a first step, we consider that a *context* Γ is composed of a set of pairs (variable,sort), and (function symbol,signature):

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

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 $\pi : \mathit{wt}$ 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*.

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

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

$$\frac{\Gamma\vdash (cond):wt \quad \Gamma\vdash e_1:s_1 \quad \ldots \quad \Gamma\vdash e_n:s_n}{\Gamma\vdash (cond\longrightarrow(e_1,\ldots,e_n)):wt}\ \text{T-RULE}$$

Figure 1: Simple type checking system.

2.1 Subtypes and associative-matching

In order to introduce subtypes in Tom, we define $\mathscr{S}$ 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.

We extend matching over lists (*i.e.* variadic operators) to be associative. Therefore a pattern matches a subject considering equality relation modulo flattening. Lists can be denoted by function symbols $l\in\mathscr{F}_v$, as said previously, 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 $l\in\mathscr{F}_v$ with variable domain (since they have a variable arity) of sort s_1 and codomain s are written $l:{s_1}^*\to s$ 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. 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 is not useful to know what the function symbol is. 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 $s_1<:s_2\wedge g_1=g_2$.

Given these notions, we define a context Γ by the following grammar:

$$\Gamma ::= \varnothing \mid \Gamma_1\cup\Gamma_2 \mid s_1^{g_1}<:_s s_2^{g_2} \mid x:s^g \mid x^*:s^l \mid f:s_1^?,\ldots,s_n^?\to s^f \mid l:(s_1^?)^*\to s^l$$

and context access is defined 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:

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

where $x\in\mathscr{X}$, $f\in\mathscr{F}$, $l\in\mathscr{F}_v$, $g,g_1,g_2\in\mathscr{F}\cup\mathscr{F}_V\cup\{?\}$ and $s^?,s_1^{g_1},s_2^{g_2}s^f,s^g,s^l\in\mathscr{D}$.

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 ones

applying 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.

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

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

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

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

$$\frac{\Gamma \vdash l(e_1, \ldots, e_n) : s^l \qquad \Gamma \vdash e : s_1^?}{\Gamma(l : (s_1^?)^* \to s^l) \vdash l(e_1, \ldots, e_n, e) : s^l}\ \text{T-ELEM}$$
if $\texttt{sortOf}(\Gamma, e) \neq s_2^l$, where $s_2^l \in \mathscr{D}$

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

$$\frac{\Gamma \vdash e : s_1^{g_1}}{\Gamma(s_1^{g_1} <:_s s^g) \vdash e : s^g}\ \text{SUB}$$
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 $\texttt{sortOf}(\Gamma, e_i) = s_i^{g_1}$, where $g_i \in \mathscr{F} \cup \mathscr{F}_v \cup \{?\}$ for $i \in [1, n]$

Figure 2: Type checking rules.

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash l() : \mathbb{Z}^l}\,\text{T-EMPTY} \quad \dfrac{}{\Gamma \vdash x^* : \mathbb{Z}^l}\,\text{T-SVAR}}{\Gamma \vdash l(x^*) : \mathbb{Z}^l}\,\text{T-MERGE} \quad \dfrac{}{\Gamma \vdash y : \mathbb{Z}^?}\,\text{T-VAR}}{\Gamma \vdash l(x^*, y) : \mathbb{Z}^l}\,\text{T-ELEM} \quad \dfrac{}{\Gamma \vdash z^* : \mathbb{Z}^l}\,\text{T-SVAR}}{\Gamma \vdash l(x^*, y, z^*) : \mathbb{Z}^l}\,\text{T-MERGE}}{\Gamma \vdash l(x^*, y, z^*) : \mathbb{Z}^?}\,\text{T-GEN} \quad \vdots \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash l() : \mathbb{Z}^l}\,\text{T-EMPTY} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash one() : \mathbb{N}^{one}}\,\text{T-FUN}}{\Gamma \vdash one() : \mathbb{N}^?}\,\text{GEN}}{\Gamma \vdash one() : \mathbb{Z}^?}\,\text{SUB}}{\Gamma \vdash l(one()) : \mathbb{Z}^l}\,\text{T-ELEM}}{\Gamma \vdash l(one()) : \mathbb{Z}^?}\,\text{T-GEN}}{\Gamma \vdash (l(x^*, y, z^*) \prec\!\!\prec_{[\mathbb{Z}^?]} l(one())) : wt}\,\text{T-MATCH} \quad \dfrac{}{\Gamma \vdash y : \mathbb{Z}^?}\,\text{T-VAR}}{\Gamma \vdash (l(x^*, y, z^*) \prec\!\!\prec_{[\mathbb{Z}^?]} l(one()) \longrightarrow (y)) : wt}\,\text{T-RULE}$$

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 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 2):

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

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:

- $\sigma\tau_1 = wt$ and $\sigma\tau_2 = wt$; or
- assuming $\sigma\tau_1 = s_1^{g_1}$ and $\sigma\tau_1 = s_2^{g_2}$, we have $s_1 <: s_2^{g_2} \wedge g_1 = g_2$, where $g_1, g_2 \in \mathscr{F} \cup \mathscr{F}_v \cup \{?\}$ and $s_1^{g_1}, s_2^{g_2} \in \mathscr{T}_{ype}(\mathscr{D} \cup \{wt\}, \mathscr{V})$;

Thus, σ satisfies 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.

Constraints are calculated according to application of rules of type inference system given in Fig. 3 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. 3 we give a type inference system with constraints. 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 $l()$ a type variable α with the constraint $\alpha = s^l$, s^l given by the signature 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* $\Gamma = \{l : (\mathbb{Z}^?)^* \rightarrow \mathbb{Z}^l, one : \rightarrow \mathbb{N}^{one}, x^* : \alpha_1, y : \alpha_2, z^* : \alpha_3, \mathbb{N}^? <:_s \mathbb{Z}^?\}$. *Then the expression* $l(x^*, y, z^*) \prec\!\!\!\prec_{[\alpha_4]} l(one()) \longrightarrow (y)$ *is well-typed and the deduction tree is:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{ct} l() : \alpha_5 \bullet \{\alpha_5 = \mathbb{Z}^l\}}\,\text{CT-EMPTY} \quad \dfrac{}{\Gamma \vdash_{ct} x^* : \alpha_{10} \bullet \{\alpha_{10} = \alpha_1\}}\,\text{CT-SVAR}}{C_2 = \{\alpha_5 = \mathbb{Z}^l\} \cup \{\alpha_{10} = \alpha_1\} \cup \{\alpha_5 = \mathbb{Z}^l, \alpha_{10} = \mathbb{Z}^l\}}\,\text{CT-STAR} \quad \dfrac{}{\Gamma \vdash_{ct} y : \alpha_9 \bullet \{\alpha_9 = \alpha_2\}}\,\text{CT-VAR}}{C_1 = C_2 \cup \{\alpha_9 = \alpha_2\} \cup \{\alpha_5 = \mathbb{Z}^l, \alpha_9 <:_s \mathbb{Z}^?\}}\,\text{CT-ELEM}}{\Gamma \vdash_{ct} l(x^*, y) : \alpha_5 \bullet C_1}}{\vdots}}{\dfrac{\dfrac{}{\Gamma \vdash_{ct} z^* : \alpha_8 \bullet \{\alpha_8 = \alpha_3\}}\,\text{CT-SVAR}}{C_p = C_1 \cup \{\alpha_8 = \alpha_3\} \cup \{\alpha_5 = \mathbb{Z}^l, \alpha_8 = \mathbb{Z}^l\}}\,\text{CT-STAR}}{\Gamma \vdash_{ct} l(x^*, y, z^*) : \alpha_5 \bullet C_p}$$

(1)

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

(2)

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

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

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

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

$$\frac{\Gamma\vdash_{ct} l(e_1,\ldots,e_n):\alpha\bullet C_1 \quad \Gamma\vdash_{ct} e:\alpha_1\bullet C_2 \quad C=C_1\cup C_2\cup\{\alpha=s^l,\alpha_1 <:_s s_1^?\}}{\Gamma(l:(s_1^?)^*\to s^l)\vdash_{ct} l(e_1,\ldots,e_n,e):\alpha\bullet C}\ \text{CT-ELEM}$$
$$\text{if } \texttt{sortOf}(\Gamma,e)\neq s_2^l \text{ and } e\neq x^*, \text{ where } s_2^l\in\mathscr{D}$$

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

$$\frac{\Gamma\vdash_{ct} l(e_1,\ldots,e_n):\alpha\bullet C_1 \quad \Gamma\vdash_{ct} x^*:\alpha_1\bullet C_2 \quad C=C_1\cup C_2\cup\{\alpha=s^l,\alpha_1=s^l\}}{\Gamma(l:(s_1^?)^*\to s^l)\vdash_{ct} l(e_1,\ldots,e_n,x^*):\alpha\bullet C}\ \text{CT-STAR}$$
$$\text{if } \texttt{sortOf}(\Gamma,x^*)\neq s^l$$

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

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

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

Figure 3: Type inference rules.

3.2 Constraint resolution

To determine if a rule expression is well-typed, its constraint set C needs to be solved in order to generate a *most general solution* σ of C from which all solutions can be generated straightforwardly. The substitution σ is said to be the *most general solution* for C if:

1. σ is a solution for C which means that $\sigma\models C$; and
2. for all solutions σ' for C, $\sigma'\alpha <:_s \sigma\alpha$ for all $\alpha\in\mathscr{V}$ in C.

The rules for the constraint resolution algorithm are provided in Fig. 4, where $g,g_1,g_2\in\mathscr{F}\cup\mathscr{F}_v\cup\{?\}$. The algorithm starts by applying closure in Γ which means it generates an assertion $s_1^{g_1} <:_s s_3^{g_3}$ in Γ

whenever $\{s_1^{g1} <:_s s_2^{g2}, s_2^{g2} <:_s s_3^{g3}\} \subseteq \Gamma$ and two other assertions $s_1^{g1} <:_s s_1^{g1}$ and $s_2^{g2} <:_s s_2^{g2}$ in Γ whenever $s_1^{g1} <:_s s_2^{g2} \in \Gamma$. Then, rules (1)-(11) 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 simplified constraints. Rules (8) and (9) treat failure. Rules (10) and (11) 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 stops if: a rule returns $C = \varnothing$, then the algorithm returns the most general solution σ; or if a rule returns $fail$ or C reaches a stable form, then the algorithm returns an error.

(1)	$\{\tau = \tau\} \uplus C', \sigma$	$\Longrightarrow$	C', σ
(2)	$\{\tau <:_s \tau\} \uplus C', \sigma$	$\Longrightarrow$	C', σ
(3)	$\{s_1^{g1} <:_s s_2^{g2}\} \uplus C', \sigma$	$\Longrightarrow$	C', σ if $s_1^{g1} <:_s s_2^{g2} \in \Gamma$
(4)	$\{\alpha = \tau\} \uplus C', \sigma$	$\Longrightarrow$	$[\alpha \mapsto \tau]C', \{\alpha \mapsto \tau\} \cup \sigma$
(5)	$\{\tau = \alpha\} \uplus C', \sigma$	$\Longrightarrow$	$[\alpha \mapsto \tau]C', \{\alpha \mapsto \tau\} \cup \sigma$
(6)	$\{\tau_1 <:_s \tau_2, \tau_2 <:_s \tau_1\} \uplus C', \sigma$	$\Longrightarrow$	$\{\tau_1 = \tau_2\} \cup C', \sigma$
(7)	$\{\alpha <:_s s_1^{g1}, \alpha <:_s s_2^{g2}\} \uplus C', \sigma$	$\Longrightarrow$	$\{\alpha <:_s min(s_1^{g1}, s_2^{g2})\} \cup C', \sigma$
(8)	$\{s_1^{g1} = s_2^{g2}\} \cup C', \sigma$	$\Longrightarrow$	$fail$ if $s_1^{g1} \neq s_2^{g2}$
(9)	$\{s_1^{g1} <:_s s_2^{g2}\} \cup C', \sigma$	$\Longrightarrow$	$fail$ if $s_1^{g1} <:_s s_2^{g2} \notin \Gamma$
(10)	$\{\alpha <:_s s^g\} \uplus C', \sigma$	$\Longrightarrow$	$[\alpha \mapsto s^g]C', \{\alpha \mapsto s^g\} \cup \sigma$ if $\alpha \notin \mathscr{V}(C')$
(11)	$\{s^g <:_s \alpha\} \uplus C', \sigma$	$\Longrightarrow$	$[s^g \mapsto \alpha]C', \{\alpha \mapsto s^g\} \cup \sigma$ if $\alpha \notin \mathscr{V}(C')$

Figure 4: Constraint resolution rules.

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

1. *Application of closure in* Γ, *generating* $\mathbb{N}^? <:_s \mathbb{N}^?$ *and* $\mathbb{Z}^? <:_s \mathbb{Z}^?$;
2. *Application of rules (1), (4) and (5) generating* $\{\mathbb{Z}^l = \mathbb{Z}^l, \mathbb{Z}^l = \mathbb{Z}^l, \mathbb{Z}^l = \mathbb{Z}^l, \mathbb{Z}^l = \mathbb{Z}^l, \alpha_2 = \alpha_2, \mathbb{Z}^l = \mathbb{Z}^l, \alpha_2 <:_s \mathbb{Z}^?, \mathbb{Z}^l = \mathbb{Z}^l, \mathbb{Z}^l = \mathbb{Z}^l, \mathbb{Z}^l = \mathbb{Z}^l, \mathbb{Z}^l = \mathbb{Z}^l, \mathbb{N}^{one} = \mathbb{N}^{one}, \alpha_6 = \mathbb{Z}^l, \mathbb{N}^{one} <:_s \mathbb{Z}^?, \mathbb{Z}^l <:_s \mathbb{Z}^l, \alpha_6 = \mathbb{Z}^l\} \cup C$ *and* $\{\alpha_5 \mapsto \mathbb{Z}^l, \alpha_{10} \mapsto \alpha_1, \alpha_1 \mapsto \mathbb{Z}^l, \alpha_9 \mapsto \alpha_2, \alpha_8 \mapsto \alpha_3, \alpha_3 \mapsto \mathbb{Z}^l, \alpha_6 \mapsto \mathbb{Z}^l, \alpha_7 \mapsto \mathbb{N}^{one}, \alpha_4 \mapsto \mathbb{Z}^l\} \cup \sigma$
3. *Application of rules (1), (2) and (3) generating* $\{\alpha_2 <:_s \mathbb{Z}^?\}$ *and* σ;
4. *Application of rule (10) 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* $l(x^*, y, z^*) \twoheadleftarrow_{[\alpha_4]} l(one()) \longrightarrow (y)$.

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* $T_1 \in \mathscr{D} \cup \{wt\}$*. Moreover, suppose that* $\Gamma \vdash e : \tau \bullet C$*. 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* $\Gamma \vdash_{ct} e : \tau \bullet C$*. If* (σ, s^g) *is a solution for* (Γ, e, τ, C)*, then it is also a solution for* (Γ, e)*.*

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

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

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

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

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

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 $\sigma\Gamma \vdash \sigma(a_1 \prec\!\!\!\prec_{[\tau_1]} a_2) : wt$, 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:
$e = l(a_1, \ldots, a_n, a)$ $\quad \tau = \alpha$
$\pi_1 = \Gamma \vdash_{ct} l(a_1, \ldots, a_n) : \alpha \bullet C_1$ $\quad \pi_2 = \Gamma \vdash_{ct} a : \alpha_1 \bullet C_2$
$C = C_1 \cup C_2 \cup \{\alpha = s_2^l, \alpha_1 <:_s s_1^?\}$ $\quad V(\pi) = \{\alpha, \alpha_1\}$
$\texttt{sortOf}(\Gamma, a) \neq s_3^l$, where $s_3^l \in \mathscr{D}$

From the assumption that (σ, s^g) is a solution for $(\Gamma(l : (s_1^?)^* \to s_2^l), l(a_1, \ldots, a_n, a))$ and $dom(\sigma) \cap V(\pi) = \varnothing$, we have $\sigma(\Gamma(l : (s_1^?)^* \to s_2^l)) \vdash \sigma(l(a_1, \ldots, a_n, a)) : s^g$. 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 $\{\alpha = s_2^l, \alpha_1 <:_s s_1^?\}$. Reason by cases as follows:

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

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

From the assumption that (σ, s^g) is a solution for $(\Gamma(l : (s_1^?)^* \to s_2^l), l(a_1, \ldots, a_n, a))$ and $dom(\sigma) \cap V(\pi) = \varnothing$, we have $\sigma(\Gamma(l : (s_1^?)^* \to s_2^l)) \vdash \sigma(l(a_1, \ldots, a_n, a)) : s^g$. 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 $\{\alpha = s_2^l\}$. Reason by cases as follows:

1. By T-MERGE we assume that $s^g = s_2^l$ and we know that $\sigma\Gamma \vdash \sigma(l(a_1, \ldots, a_n)) : s_2^l$ and $\sigma\Gamma \vdash \sigma a : s_2^l$. By the induction hypothesis, there are solutions (σ_1, s_2^l) for $(\Gamma, l(a_1, \ldots, a_n), \alpha, C_1)$ 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 $\sigma' = \{\alpha \mapsto s_2^l\} \cup \sigma \cup \sigma_1 \cup \sigma_2$. Conditions (a), (b), (c) and (d) are obviously satisfied. Thus, we see that (σ', s^g) is a solution for $(\Gamma(l : (s_1^?)^* \to s_2^l), l(a_1, \ldots, a_n, a), \alpha, C)$.

2. By T-ELEM we assume that $s^g = s_2^l$ and we know that $\sigma\Gamma \vdash \sigma(l(a_1,\ldots,a_n)) : s_2^l$ and $\sigma\Gamma \vdash \sigma a : s_1^?$. But, because of the application condition of T-ELEM, we cannot find a type s_3^l for σa such that $s_3^l <:_s s_2^l$, so $\sigma\Gamma \vdash \sigma a : s_1^?$ cannot be derived from GEN. Likewise, since we cannot find a type s_3^l for σa such that $s_3^l <:_s s_1^?$, so $\sigma\Gamma \vdash \sigma a : s_1^?$ cannot be derived even from SUB. Thus T-ELEM is not a relevant case.

3. By SUB we assume that $s_2^l <:_s s^g$ and we know that $\sigma(\Gamma(l : (s_1^?)^* \to s_2^l)) \vdash \sigma(l(a_1,\ldots,a_n,a)) : s_2^l$. This must be derived from T-MERGE, similar to case (1).

Case CT-MATCH: $e = a_1 \ \prec\!\!\!\prec_{[\tau_1]} \ a_2$ $\quad \tau = wt$
$\pi_1 = \Gamma \vdash_{ct} a_1 : \alpha_1 \bullet C_1$ $\quad \pi_2 = \Gamma \vdash_{ct} a_2 : \alpha_2 \bullet C_2$
$C = C_1 \cup C_2 \cup \{\alpha_1 <:_s \tau_1, \alpha_2 = \tau_1\}$ $\quad V(\pi) = \{\alpha_1, \alpha_2, \tau_1\}$ 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 $(\Gamma, a_1 \ \prec\!\!\!\prec_{[\tau_1]} \ a_2)$ and $dom(\sigma) \cap V(\pi) = \varnothing$, we have $\sigma\Gamma \vdash \sigma(a_1 \ \prec\!\!\!\prec_{[\tau_1]} \ a_2) : wt$. This must be derived from T-MATCH, so 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 $\{\alpha_1 <:_s \tau_1, \alpha_2 = \tau_1\}$. 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 $(\Gamma, (e = a_1 \ \prec\!\!\!\prec_{[\tau_1]} \ a_2), wt, C)$.

□

Conjecture 4.5 (Uniqueness of type)**.** *Suppose that $\Gamma \vdash_{ct} e : \tau \bullet C$. If there are two solutions $(\sigma_1, s_1^{g_1})$ and $(\sigma_2, s_2^{g_2})$ for (Γ, e, τ, C) where σ_1 and σ_2 are two most general solutions for C then $\sigma_1 = \sigma_2$ and $s_1^{g_1} = s_2^{g_2}$.*

Theorem 4.6 (Termination of algorithm)**.** *The constraint resolution algorithm always terminates, failing when given a non-satisfiable constraint set as input and otherwise returning the most general solution. More formally:*

1. *the algorithm halts, either by failing or by returning a substitution, for all C;*
2. *if the algorithm returns a σ, then σ is a solution for C;*
3. *if there exists a σ' solution for C, then the algorithm returns a σ and $\sigma'\alpha <: \sigma\alpha$ for all $\alpha \in \mathscr{V}$ in C.*

We can already sketch a proof of Theorem 4.6 following Pierce [8].

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 primitive types) 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.

For part 3, by induction on the number of recursive calls in the computation of the algorithm, reasoning by cases on the shapes of the types involved in the constraints. □

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 syntatic 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 turn the Proposition 4.5 into a theorem, which we do not expect to be too difficult. Moreover, before the implementation of the type inference algorithm, we need to have a formal proof of termination — in contrast to the sketch we currently have — as stated by Theorem 4.6.

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 [5, 6]. 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.

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

[3] 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.

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

[5] 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.

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

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

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

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