A type system for embedded rewriting languages with associative pattern matching: from theory to practice

Cláudia Tavares

A type system for embedded rewriting languages with associative pattern matching : from theory to practice

Cláudia TAVARES

November, 2011

Abstract

Programmers are often interested in a way to write error-free programs, i.e. to avoid undesired behaviors. In this context, a type system was conceived as the formal method for specification and proof of programs written in the Tom rewriting language.
The Tom programming language is an extension of Java that adds pattern matching, more particularly associative pattern matching, and reduction strategies. Accordingly, we aim to make the static typing of the Tom constructs, in particular matching constructs, compatible with the features of Java.
This paper presents a type system with nominal subtyping for Tom, that is compatible with the Java type system, and that performs both type inference. We adopted the idea that type inference is a form of type checking and included equality and subtyping constraints inside typing judgments. We propose a constraint-based type inference system to infer types of variables occurring in algebraic terms. Then, we define rules to solve equality constraints by unification. For the resolution of subtyping constraints, we introduce rules for a combination of constraint propagation, generation of solution and garbage collecting of remaining constraints.

1 Introduction

The Tom language is an extension of Java [1] that provides rule based constructs. In particular, any Java program is a Tom program. We call this kind of extension *formal islands* [1] where the *ocean* consists of Java code and the *island* of algebraic patterns. In this sense, the Tom system takes a Tom file composed of a mix between Java and Tom constructs as input and transforms it into a Java file as output. The system has a pipeline structure where each module process the given input and passes the result to the next one. In Fig. 1 we illustrate the order of modules which are responsible for different compilation phases:

Parser During parsing, an Abstract Syntax Tree (AST) is produced with special nodes for Tom constructs such as **%match** and ‘ (backquote) constructs. These nodes are Gom terms, since the Tom grammar is defined by a Gom algebraic signature. Furthermore, in case a Gom algebraic signature as (part of) the input file, the parser calls the Gom tool to produce its mapping and respective Java implementation,

Syntax checker A series of verifications on the received AST is performed at the syntax checking phase. For instance, the verification that each function symbol found was declared or that there are no circular dependencies between matching conditions,

Desugarer The desugarer fills the type mapping with constructor and destructor Java methods for each Tom constructor. This phase also simplifies the AST by transforming all different kinds of nodes representing different host language constructors into a generic node. Moreover, each anonymous variable is named with a fresh name which means a name that was never used before into the code,

1. More languages as C,C#, OCaml and Python are also supported by Tom but in this paper we consider Java as the host language code.

Typer The typer performs type inference considering types corresponding to the mappings of algebraic data types. The inferred types are propagated into the AST,

Type checker This module consists in checking the types/ranks of the symbols. For instance, it checks that two occurrences of the same variables have the same type,

Expander The expander transforms the AST in order to prepare the compilation. For instance, it generates constructs to allow the use of strategies, i.e. classes to traverse Tom terms,

Compiler The compiler locates a **%match** construct occurring in the AST and turns a rewriting rule composed of a list of several patterns into a list of several rewriting rules with a single pattern. These transformations are expressed in an intermediate language,

Optimizer This module is responsible for optimizing the code written in intermediate language. For instance, it reduces the number of assignments and tests corresponding to a **%match** construct,

Backend The backend consists in the code generation phase. During this phase, the constructs of the intermediate language are translated into host language instructions.

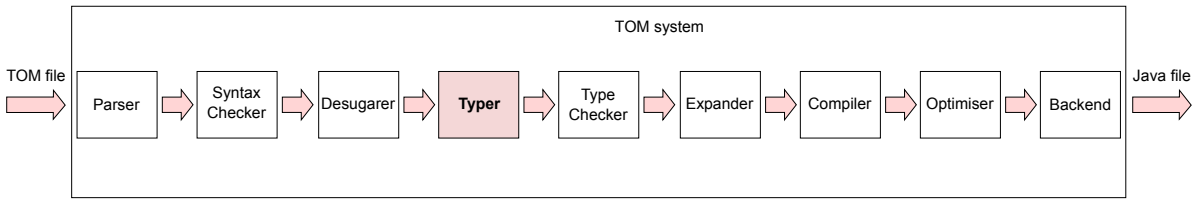

Figure 1: Modules of the TOM system.

In order to implement a typer for Tom able to infer and check types for all Tom (algebraic) terms, we define a constraint-based type system for Tom which consider the set of types as the union of Java types and abstract data types. 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 Int^+, Int^-, Int and Zero, the type system accepts the declaration $\mathsf{Int}^+ <: \mathsf{Int} \wedge \mathsf{Int}^- <: \mathsf{Int}$ but refuses the declaration $\mathsf{Zero} <: \mathsf{Int}^+ \wedge \mathsf{Zero} <: \mathsf{Int}^-$. Moreover, a function symbol f cannot be overloaded on both sorts Int^+ and Int^-.

One advantage of the constraint-based approach is that even typed terms (i.e. whose types do not need to be inferred) have their types checked since they are used by the constraints. But the mainly advantage is that the addition of new kind of constraints would only require a modification of the rules for the generation and resolution of constraints. In this sense, we divide the presentation of our type system into the version with only equality constraints and that with both equality and subtyping constraints.

In order to handle subtypes, we extend further the syntax and semantics of the Tom language by defining both a syntax for the introduction of subtypes in many-sorted algebraic signatures and a precise mapping supporting subtyping.

2 The Tom language

In a Tom program, the many-sorted algebraic signatures defined by the user must be mapped to Java classes in order to be handled into the Java program resulting. This mapping can be done either manually by the user or automatically by using the Gom program [6]. From an abstract grammar, Gom not only generates a Java implementation but also provides a mapping defining Java constructors and destructors for algebraic operators.

A many-sorted algebraic signature can be defined by a Gom grammar:

```
module Simple
  abstract syntax
  A = a()
```

```
    | f(num1:A)
    | inc1(num2:B)
    | inc2(num1:A)

  B = b()
    | g(num2:B)
    | conc1(A*)
    | conc2(B*)
```

The grammar is composed of two sorts A and B. The constructors of A are a, f, inc1 and inc2. The constructors of B are b, g, conc1 and conc2, where the last two function symbols are simultaneously associative and variadic operators and we call them *list symbols*. Considering the current grammar, we can define a well-typed **%match** parametrized by a list of rules composed of different conditions (left-hand side) and actions (right-hand side).

```
1  public void simpleMatch(A arg1, B arg2) {
2    %match {
3      x << arg1                    -> { System.out.println(`f(x)); }
4      x << arg1                    -> { System.out.println( createList (`x)); }
5      x << B arg2                  -> { System.out.println(`g(x)); }
6      conc1(x*,e) << arg2          -> { System.out.println(`e); }
7      conc2(conc1(a()),x) << arg2 -> { System.out.println("Function " + `x); }
8      conc2(conc1(a()),x*) << arg2 -> { System.out.println("List " + `x); }
9    }
10 }

12 public static B createList(A element) {
13   return `conc1(element);
14 }
```

Note that both pattern and subject are variables but do not have their sorts declared in lines 3 and 4. Their sorts are inferred by the type information of the signature of f in right-hand side. In line 4, the typer does not know the signature of createList occuring in the action side since it is not a Tomconstructor but a Java method. In this case, the sort of pattern is inferred by the sort of subject which is the same of the line 3. The subject has its sort declared in line 5 and this information is propagated to the pattern and checked when inferring the sort of the argument of g. In line 6, the signature of conc1 is used to infer sorts of x, e and arg2. Lines 7 and 8 look similar but in the first one x can be instantiated only by a function of sort B against the possibility of either a list or a function of sort B in line 8.

The typer is capable to detect when a Tom term is not well-typed while solving constraints. This includes checking of non-linearity of terms. It also checks if every algebraic types occurring in code were previously declared.

```
1  public void checkTypes(A arg1, A arg2) {
2    %match {
3      a() << C arg1 -> { System.out.println("ERROR: Undeclared sort 'C'."); }

5      x@b() << A arg1 && (f(x) == arg2) -> {
6        System.out. println ("ERROR: Incompatible sorts 'A' and 'B' for 'b' and 'x'.");
7      }

9      inc1(x) << arg1 -> {
10       System.out. println ("Out of the inner match 'x' has sort 'B'.");
11       %match {
12         inc2(x) << arg2 && (x == a()) -> {
13           System.out. println ("ERROR: Into the inner match 'x' has sort 'A'.");
14         }
15       }
16     }
17   }
18 }
```

In line 3, the typer finds an unknown sort and points out an error since sort C was not declared in the Gom grammar. Since the sort of the signature of b is different from that assigned by the subject of the matching in line 5, an error of incompatible sorts is indicated. Moreover, the same error happens to the variable x occurring twice in the left-hand side. x is typed with both the sort of the signature of

b and the sort of the domain of f. Incompatible sorts are also assigned to another variable x occurring non-linearly in the condition of line 9 and that of the embedded **%match** in line 12.

In order to define a grammar considering subtypes, a many-sorted algebraic signature can be manually defined by providing a Java implementation for each Tom sorts and operators.

```
class A {
  public A() {}
  public String getOp() { return ""; }
}

class Javaa extends A {
  public Javaa() { }
  public String getOp() { return "a"; }
}

class Javaf extends A {
  public A num1;
  public Javaf(A n1) { num1 = n1; }
  public A getnum1() { return num1; }
  public String getOp() { return "f"; }
}

class B extends A {
  public B() {}
  public String getOp() { return ""; }
}

class Javab extends B {
  public Javab() { }
  public String getOp() { return "b"; }
}

class Javag extends B {
  public B num2;
  public Javag(B n2) { num2 = n2; }
  public B getnum2() { return num2; }
  public String getOp() { return "g"; }
}
```

```
%typeterm TomA {
  implement { A }
  is_sort (t) { $t instanceof A }
  equals(t1,t2) { ($t1==$t2) }
}

%typeterm TomB extends TomA {
  implement { B }
  is_sort (t) { $t instanceof B }
  equals(t1,t2) { ($t1==$t2) }
}

%op TomA a() {
  is_fsym(t) { $t instanceof Javaa }
}

%op TomA f(num1:TomA) {
  is_fsym(t) { $t instanceof Javaf }
  get_slot(num1, t) { ((Javaf)$t).getnum1() }
}

%op TomB b() {
  is_fsym(t) { $t instanceof Javab }
}

%op TomB g(num2:TomB) {
  is_fsym(t) { $t instanceof Javag }
  get_slot(num2, t) { ((Javag)$t).getnum2() }
}
```

The mapping defines a sort A having a constant constructor a and a function constructor f which takes an argument of sort A. Another sort B is defined as a subtype of A and has a constant constructor b and a function constructor g which takes an argument of sort B.

Despite the possibility for definition of a partially ordered set of sorts, Tom is not able to know the order relation between two given sorts. This happens because the relationships are defined by Java constructs instead of Tom constructs. The introduction of subtypes into Tom typer will allow more flexibility when coding Tom programs.

```
1  public final static void main(String[] args) {
2    SubtypeExample test = new SubtypeExample();
3    test.buildExpA();
4    test.buildExpB();
5  }

7  public void buildExpA() {
8    print(new Javaf(new Javaa()));
9  }

11 public void buildExpB() {
12   print(new Javag(new Javab()));
13 }

15 public void print(A term) {
16   String op = term.getOp();
17   System.out.print("Term = " + `op);
18   %match {
19     f(arg) << TomA term -> { System.out.println("(" + `arg.getOp() + ")"); }
20     g(arg) << TomB term -> { System.out.println("(" + `arg.getOp() + ")"); }
21   }
22 }
```

Note that a subject of sort A can be matched against the patterns of the same sort A and those of a subsort B by doing explicit downcast as in line 20. However Tom currently offers nothing to note a relation between sorts. Such subtyping annotations calls for a proper extension of Tom syntax. This extension in turn affects Tom typer. We started the process bottom-up by implementing the typer which will support those new constructs.

3 The Tom syntax

The current version of the typer was implemented using the Tom language and the Tom system was successfully bootstrapped. In order to present Tom syntax we introduce the notion of sorts decorated with function symbols to classify terms, for instance s^g with $\mathsf{g} \in \mathcal{F} \cup \mathcal{F}^\star$. We also define a special symbol ? for the cases in which the decoration is unimportant, for example, when specifying the domain of a function symbol. Decorated sorts are also called *ground types* since they are the simplest types with whom a term can be classified. We introduce a special ground type wt to classify expressions which are not terms, for instance matching conditions, backquote constructs among others. On the whole, the set of types $\mathcal{T}y$, is composed of unsorted terms built from a set of *type variables*, from $\{wt\}$ and from the set of decorated sorts, i.e. the combination of $\mathcal{S}$ and $\mathcal{F} \cup \mathcal{F}^\star \cup \{?\}$.

Definition 3.1 (Types)**.** *Let wt be a special constant and $\mathcal{X}$ be a countably infinite set of* type variables *denoted by α, β, etc. The set $\mathcal{T}y$ of types is made up of the unsorted terms defined by the following algebraic grammar:*

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

where $\tau \in \mathcal{T}y$, $s \in \mathcal{S}$, $h \in \mathcal{F} \cup \mathcal{F}^\star \cup \{?\}$.

A ground type *is a type τ with no variables, i.e. $\mathcal{V}ar(\tau) = \emptyset$.*

The equality of decorated sorts consists in the syntactic equality of sorts enriched by a special comparison of their decorations.

Definition 3.2 (Equality of decorated sorts)**.** *Consider the set $\mathcal{T}y$ of types, i.e. decorated sorts.* Equality of decorated sorts*, denoted by $=_t$, is defined by equality modulo the following property:*

$$\forall s_1^{h_1}, s_2^{h_2} \in \mathcal{T}yn(\mathcal{X} \cup \{wt\}), (s_1^{h_1} =_t s_2^{h_2} \Leftrightarrow (s_1 = s_2 \wedge (h_1 = h_2 \vee h_1 \ = ? \vee h_2 \ = ?)))$$

We write $s_1^{h_1} \neq_t s_2^{h_2}$ to mean that $\neg(s_1^{h_1} =_t s_2^{h_2})$.

The notion of types is really useful for the typing of Tom expressions. For this reason, we define a correspondence between types and Tom algebraic data types, referred to as base sorts: the partial function of *decoration cleaning* allows types to be converted into non-decorated sorts. We mainly use this when representing the explicit type annotation of the **%match** construct through a type.

Definition 3.3 (Decoration cleaning)**.** *The function $|\cdot|$ is a partial function from $\mathcal{T}y$ to $\mathcal{S}$ defined by:*

$$|s^h| = s$$

where $s \in \mathcal{S}$ and $h \in \mathcal{F} \cup \mathcal{F}^\star \cup \{?\}$.

Considering the set $\mathcal{T}y$ of types, the sets $\mathcal{F}$ and $\mathcal{F}^\star$ of function symbols and a countably infinite set $\mathcal{V}$ of variables, Tom terms are built from the term algebra $\mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$. The set of Tom ground terms is denoted by $\mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star)$.

Definition 3.4 (Tom terms)**.** *Tom terms are specified by the following algebraic grammar:*

$$term ::= \mathsf{x} \mid \mathsf{x*} \mid \mathsf{g}(term,\ldots,term)$$

where $\mathsf{x} \in \mathcal{V}$ and $\mathsf{g} \in \mathcal{F} \cup \mathcal{F}^\star$.

A Tom code can contains Java code which in its turn can contains Tom code and so on. Since we intend to check and infer types of Tom terms, we consider an abstraction consisting only of expressions involving Tom terms. Thus, Java variables bounded to Tom terms are represented by variables of $\mathcal{V}$. In addition, Java functions manipulating Tom terms or being manipulated by Tom terms are entirely represented by Tom terms. The remaining Java objects are dismissed.

Definition 3.5 (Tom core abstract syntax)**.** *The* Tom core abstract syntax *is specified by the following algebraic grammar:*

$$\begin{array}{lcl} \mathit{code} & ::= & \{\mathit{rule};\ldots;\mathit{rule}\} \mid \text{`}\mathit{term} \mid \mathsf{x} := \text{`}\mathit{term} \\ \mathit{rule} & ::= & \mathit{cond} \longrightarrow \mathit{action} \\ \mathit{cond} & ::= & \mathit{pattern} \prec\!\!| \tau | \ \mathit{term} \mid \mathit{term} \diamond \mathit{term} \mid \mathit{cond} \wedge \mathit{cond} \mid \mathit{cond} \vee \mathit{cond} \\ \mathit{pattern} & ::= & \mathsf{x} \mid \mathsf{x}^* \mid \mathsf{g}(\mathit{pattern},\ldots,\mathit{pattern}) \mid \mathsf{x}@\mathit{pattern} \mid !\mathit{pattern} \\ \mathit{term} & ::= & \mathsf{x} \mid \mathsf{x}^* \mid \mathsf{g}(\mathit{term},\ldots,\mathit{term}) \\ \mathit{action} & ::= & \mathit{code} \mid (\mathit{action};\ldots;\mathit{action}) \end{array}$$

where $\mathsf{x} \in \mathcal{V}$, $\mathsf{g} \in \mathcal{F} \cup \mathcal{F}^\star$ *and* $\tau \in \mathcal{T}yn\{wt\}$.

Terms built from this grammar are called Tom expressions.

A Tom code is a set of code.

A *code* can be an assignment instruction, a backquoted term or a list of rules $\mathit{cond} \longrightarrow \mathit{action}$. The assignment of a term t to a variable x is represented by $\mathsf{x} := \text{`}\mathsf{t}$. The left-hand side of a rule is either a single condition or a conjunction/disjunction of single conditions. A single condition is either a *matching* or a *numeric condition*. *Matching conditions* are match-equations $\mathsf{t}_1 \prec\!\!| \tau | \ \mathsf{t}_2$. τ denotes a type and $|\tau|$ is its undecorated version (see Definition 3.3). We call *anti-pattern* the patterns composed of one or more $!\mathit{pattern}$. *Numeric conditions* are conditions of the form $\mathsf{t}_1 \diamond \mathsf{t}_2$ where $\diamond$ stands for the representation of the logical operators $=$, $\neq$, $<$, $\leq$, $>$ and $\geq$. The right-hand side of a rule is possibly composed of one or more *code*. Thus, an action can be a sequence of actions, i.e. a sequence of list of rules and/or backquoted terms. A $\text{`}\mathit{term}$ corresponds to the backquote construct allowing a Tom term to be built into a block of Java code. Moreover, anonymous variables _ do not appear in the grammar since they are represented by *fresh* variables whose names were never used before.

Example 3.6. *The block of the* print *method given above is represented by the following* Tom *code:*

$$\{\mathsf{g(arg)} \prec\!\!| \mathit{TomA} | \ \mathsf{term} \longrightarrow \text{`}\mathsf{arg};\ \mathsf{f(arg)} \prec\!\!| \mathit{TomB} | \ \mathsf{term} \longrightarrow \text{`}\mathsf{arg}\ \}$$

4 A constraint-based type inference system

In practice, a signature $\mathcal{F} \cup \mathcal{F}^\star$ and a set of types $\mathcal{T}y$ are obtained from the abstract grammar declared by the user. All function symbols and types are respectively kept into a symbol table and type table available for all Tommodules. During the parsing, list symbols have theirs sorts enriched by theirs corresponding function symbols and added to the type table. These decorated sorts are represented by types s^v where $s \in \mathcal{T}yn(\mathcal{X} \cup \{wt\})$ and $\mathsf{v} \in \mathcal{F}^\star$. Since the decoration is not relevant for the remaining functions (i.e. non-variadic and non-associative functions) theirs sorts are directly added to the type table without additional information. These sorts are represented by types $s^?$.

Into the symbol table, functions symbols having unknown types are typed with fresh type variables. By "fresh type variable" we mean a type variable having a name that was never used before. With respect to variables, before start the type inference of a block, all variables occurring in it are also typed using fresh type variables, even those variables having the same name. The Tom typer starts the process of typing by considering each block separately in order to collect all type constraints into a constraint set $\mathcal{C}$. Each type variable introduced in a sub-derivation is a fresh type variable and fresh type variables in different sub-derivations are distinct.

4.1 Equality constraints

Both symbol table and type table deal with a *context* defined as a set of pairs (variable,type) and (operator,rank).

Definition 4.1 (Context)**.** *A* Context *is defined by:*

$$\Gamma ::= \varnothing \mid \Gamma; \mathsf{x} : \tau \mid \Gamma; \mathsf{f} : s_1^?, \ldots, s_n^? \to s^? \mid \Gamma; \mathsf{v} : (s_1^?)^* \to s^v$$

where $\mathsf{x} \in \mathcal{V}$*,* $\mathsf{f} \in \mathcal{F}$*,* $\mathsf{v} \in \mathcal{F}^\star$*,* $\tau \in \mathcal{T}y$ *and* $s^?, s_i^?, s^v \in \mathcal{T}yn(\mathcal{X} \cup \{wt\})$*. Moreover, variadic operators* v *having an indefinite arity with domain* $(s_1^?)^\star$ *and codomain* s^v *are written* $\mathsf{v} : (s_1^?)^* \to s^v$*, according to Definition* **??**.

A context Γ associates types to variables. It also associates ranks to either syntactic or variadic operators. We denote by $\Gamma(\gamma)$ the fact that a pair γ belongs to Γ. The context has at most one declaration of type per variable and of rank per operator since operator overloading is forbidden. The initialization of the context occurs during parsing: each variable of a (matching or numeric) condition is paired with its sort decorated with ? while each algebraic operator of a Tom mapping is paired with their rank. The ranks of algebraic operators are built from decoration of their sorts in the following way:

- for a syntactic operator f: all sorts appearing in the rank are decorated with ?,
- for a variadic operator v: all sorts appearing in the domain are decorated with ? while the sort of the codomain is decorated with v.

In fact, Tom ignores Java code, but for the purpose of verifying types Tom terms, we consider that the rank of Java operators and types of Java variables are known. Consequently, Java operators (represented by Tom operators) and their ranks as well as Java variables and their types are also declared into the context. Furthermore, context access is defined by the function typeof(Γ,t) which returns the type of term t in the context Γ.

Definition 4.2 (Context access)**.** *The* context access *is done by a partial binary function* typeof : $\Gamma \times \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V}) \to \mathcal{T}y$ *defined by:*

typeof(Γ,x) = τ, *if* x : $\tau \in \Gamma$
typeof(Γ,x@t) = typeof(Γ,t)
typeof(Γ,!t) = typeof(Γ,t)
typeof(Γ,f(t$_1$,...,t$_n$)) = $s^?$, *if* f : $s_1^?, \ldots, s_n^? \to s^? \in \Gamma$
typeof(Γ,v(t$_1$,...,t$_n$)) = s^v, *if* v : $(s_1^?)^* \to s^v \in \Gamma$

where $\mathsf{x} \in \mathcal{V}$*,* $\mathsf{f} \in \mathcal{F}$*,* $\mathsf{v} \in \mathcal{F}^\star$*,* $s^?, s_i^?, s^v \in \mathcal{T}yn(\mathcal{X} \cup \{wt\})$ *and* $\mathsf{t}, \mathsf{t}_i \in \mathcal{T}(\mathcal{T}y, \mathcal{F} \cup \mathcal{F}^\star, \mathcal{V})$ *for* $i \in [1, n]$.

In order to collect all type constraints generated during the typing process, we use a set of *constraints*. Thus, these constraints limit types and consequently base sorts (i.e. algebraic data types) that terms can have.

Definition 4.3 (Constraint set)**.** *Consider the set* $\mathcal{T}y$ *of types. A* constraint set $\mathcal{C}$ *is a set of constraints defined by the following algebraic grammar:*

$$c ::= \tau_1 =_t \tau_2$$

where $c \in \mathcal{C}$ *and* $\tau_1, \tau_2 \in \mathcal{T}y$*.*

A ground constraint *is a constraint with no variables, i.e. a constraint built from ground types.*

A substitution σ *is said to* satisfy *an equation* $\tau_1 =_t \tau_2$ *if* $\sigma\tau_1 =_t \sigma\tau_2$*. Thus,* σ *satisfies* $\mathcal{C}$ *if it satisfies all equations in* $\mathcal{C}$*. This is written* $\sigma \models \mathcal{C}$*.*

The set $\mathcal{X}(\mathcal{C})$ *denotes the set of type variables existing in* $\mathcal{C}$*.*

Based on unification, we define a *solution* and a *principal solution* for a constraint set $\mathcal{C}$.

Definition 4.4 (Solution for constraint set)**.** *Let* $\mathcal{C}$ *be a constraint set composed of equality constraints* $\tau_1 =_t \tau_2$*. A* solution *of an equality constraint* $\tau_1 =_t \tau_2$ *is a substitution* σ *such that* $\sigma\tau_1 =_t \sigma\tau_1$*. A substitution is solution of* $\mathcal{C}$ *if it is solution of every equality constraint in* $\mathcal{C}$*.*

A constraint typing judgment for Tom expressions has the form $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$ and is defined by a set of inference rules assigning types to Tom expressions, summarized in Fig. 2. Constraints are calculated according to the application of these rules. Informally, $\Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$ can be read "the expression e is of type τ under assumptions Γ whenever the constraints $\mathcal{C}$ are satisfied". Formally, the typing judgment states that

$$\forall \sigma(\sigma \models \mathcal{C} \Rightarrow \sigma\Gamma \vdash \sigma e : \sigma\tau)$$

Derivations of the type inference algorithm are read bottom-up and implemented by a strategy. The strategy traverses a Tom program and tries to apply one of the derivations, collecting constraints into a set $\mathcal{C}$. The most interesting rules are those that apply to lists. For a non-empty list $\mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n)$ the rule [CT-Elem] is applicable. Therefore, if the element t_n is of type of the domain of v and its head symbol is different from v, then a type α such that $\alpha =_t s^v$ is a type for the original list incremented by t_n. It is still possible to concatenate two lists by application of the rule [CT-Merge]. If these lists are of the same type s^v then a type α such that $\alpha =_t s^v$ is a type for the resulting list. The rule [CT-Merge] is also applied to insert a star variable of type s^v into a list with codomain s^v.

We note that the rule [CT-Block] works with a union of a global context (common to all typing judgments in the premise) and local contexts Γ_i for each $rule_i$. A local context Γ_i contains all variables occurring in *pure* algebraic terms of a Tom program, i.e. in numeric conditions and in patterns of matching conditions of $rule_i$ with their respective annotations. This avoid the need of renaming variables with same name but occurring in different rules $cond \longrightarrow action$. The union $\Gamma \cup \Gamma_i$ comprises all annotated variables and operators occurring in a $rule_i$ expression.

Example 4.5. *Let* $\Gamma = \Gamma' \cup \Gamma_1$ *such that*

$$\Gamma' = (\mathsf{suc}: \ Nat^? \to Nat^?, \mathsf{concNat} : (Nat^?)^* \to NatList^{concNat}, \mathsf{nList} : \alpha_1)$$

and

$$\Gamma_1 = (\ \mathsf{x} : \alpha_2, \mathsf{y} : \alpha_3, \mathsf{z} : \alpha_4, \mathsf{pnat} : \alpha_5)$$

Then, an expression

$$\{\mathsf{concNat(x*,pnat@suc(y),z*)} \twoheadleftarrow |NatList^?| \ \mathsf{nList} \longrightarrow \mathsf{pnat}\}$$

can be deduced by the tree given in Fig. 3, where a constraint set $\mathcal{C}$ *is generated.*

4.2 Subtyping constraints

In order to introduce subtypes in Tom, we refine $\mathcal{S}$ with the addition of a partial order $<:$ over sorts called *subtyping*. It is a binary relation on $\mathcal{S}$ that satisfies reflexivity, transitivity and antisymmetry. Then we extend the notion of *subtyping* over the set of types $\mathcal{T}y$ composed of decorated sorts (see Definition 3.1). This leads to $\mathcal{T}y$ equipped with another partial order $<:_t$.

Definition 4.6 (Subtyping over decorated sorts)**.** *Consider the set* $\mathcal{T}y$ *of types, i.e. decorated sorts.* Subtyping over decorated sorts*, denoted by* $<:_t$*, is a partial order (see Definition* **??***) over* $\mathcal{T}y$ *defined by subtyping modulo the following property:*

$$\forall s_1^{h_1}, s_2^{h_2} \in \mathcal{T}yn(\mathcal{X} \cup \{wt\}) \,.\, (s_1^{h_1} <:_t s_2^{h_2} \Leftrightarrow (s_1 <: s_2 \wedge (h_1 = h_2 \vee h_2 \ = ?)))$$

In our type system, types are interpreted as unsorted terms and subtyping is nominal. Thus, subtype relations must be explicitly declared, although multiple inheritance and overloading of operators are forbidden. For instance, given the types $\mathsf{Neg}^?$, $\mathsf{ZNat}^?$, $\mathsf{Nat}^?$ and $\mathsf{Int}^?$, the type system accepts the declaration $\mathsf{Neg}^? <:_t \mathsf{Int}^? \wedge \mathsf{Nat}^? <:_t \mathsf{Int}^?$ but refuses the declaration $\mathsf{ZNat}^? <:_t \mathsf{Neg}^? \wedge \mathsf{ZNat}^? <:_t \mathsf{Nat}^?$. Moreover, a function symbol suc cannot be overloaded on both types $\mathsf{Neg}^?$ and $\mathsf{Nat}^?$.

We refine the notion of context Γ with a new kind of pairs (type,type) in addition to the pairs (variable,type) and (operator,rank) previously introduced in Definition 4.1.

Definition 4.7 (Context with subtypes)**.** *A* Context *is defined by:*

$$\Gamma ::= \varnothing \mid \Gamma; s_1^{h_1} <:_t s_2^{h_2} \mid \Gamma; \mathsf{x} : \tau \mid \Gamma; \mathsf{f} : s_1^?, \ldots, s_n^? \to s^? \mid \Gamma; \mathsf{v} : (s_1^?)^* \to s^v$$

where $s_1^{h_1}, s_2^{h_2} \in \mathcal{T}yn(\mathcal{X} \cup \{wt\})$ *.*

The access to the context is possible by the use of the binary function typeof as in Definition 4.2. The application of the transitive closure of $<:_t$ to all subtyping declarations found in Γ generates Γ^*.

A common approach to perform type inference with subtyping is based on limiting type variables through subtyping constraints in order to find a type substitution satisfying these constraints. In this sense, we must extend the notion of *constraint set* and *solution of constraint set* initially dealing with equality constraints as introduced in Definitions 4.3 and 4.4.

Definition 4.8 (Constraint set)**.** *Consider the set* $\mathcal{T}y$ *of types. A* constraint set $\mathcal{C}$ *is a set of constraints defined by the following algebraic grammar:*

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

where $c \in \mathcal{C}$*,* $\tau_1, \tau_2 \in \mathcal{T}y$*.*

A ground constraint *is a constraint with no variables, i.e. a constraint built from ground types.*

A substitution σ *is said to* satisfy *an equation* $\tau_1 =_t \tau_2$ *if* $\sigma(\tau_1) =_t \sigma(\tau_2)$*. Similarly,* σ *is said to* satisfy *a subtyping constraint* $\tau_1 <:_t \tau_2$ *if* $\sigma(\tau_1) <:_t \sigma(\tau_2)$*. Thus,* σ *satisfies* $\mathcal{C}$ *if it satisfies all constraints in* $\mathcal{C}$*. This is written* $\sigma \models \mathcal{C}$*.*

The set $\mathcal{X}(\mathcal{C})$ *denotes the set of type variables existing in* $\mathcal{C}$*.*

Definition 4.9 (Solution for constraint set)**.** *Let* $\mathcal{C}$ *be a constraint set composed of subtyping constraints* $\tau_1 <:_t \tau_2$*. A* solution *of a subtyping constraint* $\tau_1 <:_t \tau_2$ *is a substitution* σ *such that* $\sigma\tau_1 <:_t \sigma\tau_1$*. A substitution* σ *is solution of* $\mathcal{C}$ *if it is solution of every subtyping constraint in* $\mathcal{C}$*.*

The type inference rules considering subtypes are presented in Fig. 4.

As in Section 4.1, the derivations are read bottom-up and implemented by a strategy in order to collect constraints into a set $\mathcal{C}$. The strategy traverses a Tom program and tries to apply one of the derivations until reach the end of a block.

Example 4.10. *Let* $\Gamma_{init} = \Gamma' \cup \Gamma_1$ *such that*

$$\Gamma' = (\mathsf{zero} : \ ZNat^?, \mathsf{concNat} : (Nat^?)^* \to NatList^{concNat}, \mathsf{nList} : \alpha_1)$$

and

$$\Gamma_1 = (\ \mathsf{x} : \alpha_2, \mathsf{z} : \alpha_3, \mathsf{pnat} : \alpha_4)$$

Let $\Gamma = \Gamma^*_{init}$*. Consider the expression*

$$\{\mathsf{concNat(x*,pnat@!zero(),z*)} \nprec |NatList^?| \ \mathsf{nList} \longrightarrow \text{`}\mathsf{pnat}\}$$

which uses subtyping and is extensionally equivalent to the expression in Example 4.5

$$\{\mathsf{concNat(x*,pnat@suc(y),z*)} \nprec |NatList^?| \ \mathsf{nList} \longrightarrow \text{`}\mathsf{pnat}\}$$

It can be deduced by the tree given in Fig. 5, where a constraint set $\mathcal{C}$ *is generated.*

5 Equality constraint resolution

We have done the generation of equality constraints by application of type inference rules. In this section we shall focus on how to solve equality constraints. First and foremost, we consider the initial equality constraint set in *canonical form.*

Definition 5.1 (Canonical form of equality constraint set)**.** *An equality constraint set $\mathcal{C}$ is said to be in* canonical form *if it satisfies de following property:*

$\forall \alpha \in \mathcal{X}, \forall s^? \in \mathcal{T}yn(\mathcal{X} \cup \{wt\}), \forall v \in \mathcal{F}^\star$
$(\alpha =_t s^v \in \mathcal{C} \Rightarrow \alpha =_t s^? \notin \mathcal{C})$

In Fig. 6 we present a constraint resolution algorithm solveEqConstraints composed of a set of conditional rewrite rules. It produces a set *err* of type errors possibly found and a substitution σ that satisfies an initial input constraint set. The algorithm is based on the idea, due to the independent works of Hindley [2] and Milner [3], of using unification to check that a constraint set has a solution and, if so, to find a *principal solution* to it.

Definition 5.2 (Principal solution for constraint typing judgment)**.** *Let Γ be a context and e a Tom expression. Suppose that $\Gamma \vdash_{ct} \mathsf{e} : \tau \bullet \mathcal{C}$. A* principal solution *for $(\Gamma, e, \tau, \mathcal{C})$ is a solution (σ, τ') such that, whenever (σ_1, τ_1) is also a solution for $(\Gamma, e, \tau, \mathcal{C})$, we have $\sigma \lesssim \sigma_1$ (see Definition* **??***).*

A principal solution (σ, τ) for $(\Gamma, e, \tau, \mathcal{C})$ is called a principal type *of e under Γ.*

The rules of Fig. 6 are applied to all elements of a constraint set $\mathcal{C}$ to perform unification. It considers the commutative property of the $=_t$ operator. For an equation of the form $\tau_1 =_t \tau_2$, each possible instances of τ_1 and τ_2 are considered. The auxiliary algorithm isEq presented in Fig. 7 is called each time a ground equation is found in order to verify if the equality of decorated sorts (as described in Definition 3.2) holds for τ_1 and τ_2. In case the constraint is not ground, then σ is incremented according to the possible type assigned to τ_1 and τ_2 by σ.

Initially, $\mathcal{C}$ has only equations with two types τ_1 and τ_2 and the solution σ is empty. In order to keep a track of type errors found during constraint resolution, the rules increment a set *err*, initially empty, of pairs of types for which the equality of decorated sorts does not hold. Each element of *err* produces an error message and states that $\mathcal{C}$ has no solution, i.e. that the solution σ generated is not valid. This approach of collection of pairs of types corresponds to a Java-like approach since it allows the type system to raise all type errors found during constraint resolution (instead of only the first one). Nevertheless, another possible approach would be to stop the constraint resolution when *err* becomes non-empty for the first time, i.e. to add a condition as $\mathsf{isEq}(\tau_1, \tau_2) \neq \emptyset$ for each rule.

The operational behavior of the constraint resolution consists in folding the algorithm of Fig. 6 over the initial constraint set $\mathcal{C}$ with an empty set *err* of type errors and an empty solution σ:

foldl solveEqConstraints (err, σ) $\mathcal{C}$

The algorithm solveEqConstraints is non-recursive and corresponds to a derecursivated version of the classical syntactic unification algorithm, except for the manipulation of a set *err* of type errors. At every step, it takes the first equation of $\mathcal{C}$ and tries to apply one of the conditional rewrite rules and then takes the second equation to do the same and so forth. The application of σ to the current equation is done indirectly by conserving σ in a *saturated form.*

Definition 5.3 (Saturated form)**.** *Let Γ be a context, e a Tom expression and σ a solution for $(\Gamma, e, \tau, \mathcal{C})$. σ is said to be in* saturated form*, denoted by $\downarrow\sigma$, if it satisfies the following property:*

$$\forall \alpha \in \mathcal{X} (\alpha \in \mathcal{D}om(\sigma) \Rightarrow \alpha \notin \mathcal{R}ange(\sigma))$$

The following pseudocode ensures that the addition of a pair (Key,Value) keeps Substitution in saturated form:

```
GET Key and Value of pair to be added to Substitution
IF Value is in domain of Substitution THEN
    Add pair (Key, Substitution [Value]) to Substitution
ELSE
    Add pair (Key,Value) to Substitution
ENDIF

IF Key is in range of Substituion THEN
    FOR each CurrentKey in domain of Substitution
        IF Substitution [CurrentKey] is equal to Key THEN
            Overwrite pair (CurrentKey, Substitution [Key]) in Substitution
        ENDIF
    ENDLOOP
ENDIF
```

The algorithm stops when the end of the constraint set is reached. If $err \neq \emptyset$, then all errors messages collected during the resolution have been raised and the algorithm fails since σ is rejected. Otherwise, σ is applied to the Tom expression considered in the constraint typing judgment.

Example 5.4. *Considering Example 4.5, in the constraint typing judgment*

$$\Gamma \vdash_{ct} \{\text{concNat(x*,pnat@suc(y),z*)} \prec\!| NatList^{?} |\ \text{nList} \longrightarrow \text{pnat}\} : wt \bullet \mathcal{C}$$

the generated constraint set in canonical form is $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3$ *where*

$\mathcal{C}_1 = \{\alpha_6 =_t \alpha_7, \alpha_6 =_t NatList^{concNat}, \alpha_6 =_t NatList^{concNat}, \alpha_8 =_t Nat^{?}\}$
$\mathcal{C}_2 = \{\alpha_6 =_t NatList^{concNat}, \alpha_6 =_t NatList^{concNat}, \alpha_2 =_t \alpha_6, \alpha_5 =_t \alpha_8, \alpha_8 =_t Nat^{?}\}$
$\mathcal{C}_3 = \{\alpha_9 =_t Nat^{?}, \alpha_3 =_t \alpha_9, \alpha_4 =_t \alpha_6, \alpha_1 =_t \alpha_7, \alpha_5 =_t \alpha_{10}\}$

1. *Let* $err_1 = \{\}$ *and* $\sigma_1 = \{\}$. *Applying* solveEqConstraints *to* $(\mathcal{C}_1, err_1, \sigma_1)$ *with the sequence of rules* $(4a), (3c), (3b)and(3a)$, *we obtain* $(\{\}, err_1, \{\alpha_6 \mapsto NatList^{concNat}, \alpha_7 \mapsto NatList^{concNat}, \alpha_8 \mapsto Nat^{?}\} \circ \sigma_1)$;
2. *Let* $err_2 = \{\}$ *and* $\sigma_2 = \{\alpha_6 \mapsto NatList^{concNat}, \alpha_7 \mapsto NatList^{concNat}, \alpha_8 \mapsto Nat^{?}\}$. *Applying* solveEqConstraints *to* $(\mathcal{C}_2, err_2, \sigma_2)$ *with the sequence of rules* $(3b), (3b), (4b), (4b)and(3b)$, *we obtain* $(\{\}, err_2, \{\alpha_2 \mapsto NatList^{concNat}, \alpha_5 \mapsto Nat^{?}\} \circ \sigma_2)$;
3. *Let* $err_3 = \{\}$ *and* $\sigma_3 = \{\alpha_6 \mapsto NatList^{concNat}, \alpha_7 \mapsto NatList^{concNat}, \alpha_8 \mapsto Nat^{?}, \alpha_2 \mapsto NatList^{concNat}, \alpha_5 \mapsto Nat^{?}\}$. *Applying* solveEqConstraints *to* $(\mathcal{C}_3, err_3, \sigma_3)$ *with the sequence of rules* $(3a), (4b), (4b), (4b)and(4b)$, *we obtain* $(\{\}, err_3, \{\alpha_9 \mapsto Nat^{?}, \alpha_3 \mapsto Nat^{?}, \alpha_4 \mapsto NatList^{concNat}, \alpha_1 \mapsto NatList^{concNat}\}, \alpha_{10} \mapsto Nat^{?}\} \circ \sigma_3)$.

The constraint set $\mathcal{C}$ *is empty and then the algorithm stops. Moreover,* $\{\alpha_1 \mapsto NatList^{concNat}, \alpha_2 \mapsto NatList^{concNat}, \alpha_3 \mapsto Nat^{?}, \alpha_4 \mapsto NatList^{concNat}, \alpha_5 \mapsto Nat^{?}, \alpha_6 \mapsto NatList^{concNat}, \alpha_7 \mapsto NatList^{concNat}, \alpha_8 \mapsto Nat^{?}, \alpha_9 \mapsto Nat^{?}, \alpha_{10} \mapsto Nat^{?}\}$ *is generated as a solution for* $(\Gamma, \{\text{concNat(x*,pnat@suc(y),z*)} \prec\!| NatList^{?} |\ \text{nList} \longrightarrow \text{`pnat}\}, wt, \mathcal{C})$, *since* $err_1 \cup err_2 \cup err_3 \cup err_4 = \emptyset$.

6 Subtyping constraints resolution

The type inference rules describe an algorithmic way to map types of terms occurring in an expression to a constraint set. When considering subtyping, the constraint set obtained can be divided into two subsets: an equality constraint subset and a subtyping constraint subset. The former subset is solved by unification through algorithm solveEqConstraints, generating a substitution set σ in saturated form for type variables. If none errors were found during the unification (i.e. $err = \emptyset$), then the substitution is applied to the subtyping constraint subset. Otherwise, errors are raised out and the constraint resolution stops. In this section we describe a simplification algorithm inspired by the work of François Pottier [5, 4] to propagate subtyping constraints. It is done by the combination of three simplification phases which keeps σ in saturated form. The constraint propagation of subtyping constraints is described in the following and afterwards we present a phase responsible for the generation of *solution*. The last phase is the one for garbage collection on possible remaining constraints. As in equality constraint resolution, the solution σ and the set err of errors are initially empty. Moreover, $\mathcal{C}$ has only equality or subtyping constraints between types τ_1 and τ_2. However, a non-empty set err indicates that either $\mathcal{C}$ has no solution or a solution for $\mathcal{C}$ could not be found.

6.1 Simplification and closure

This phase performs a trivial elimination of reflexive subtyping constraints. They are considered meaningless since they do not add information about types restricted by them. Thus, these constraints are removed from the subtyping constraint set $\mathcal{C}$:

$$\{\tau <:_t \tau\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad \mathcal{C}', err, \sigma$$

where $\tau \in \mathcal{T}yn\{wt\}$.

The next property of the partial order $<:_t$ to be considered is the antisymmetric one. In order to solve a highest number of constraints by unification, we try to rewrite subtyping constraints into equality constraints as much as possible. For this purpose, we consider the following simplification rule:

$$\{\tau_1 <:_t \tau_2, \tau_2 <:_t \tau_1\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad \mathcal{C}', \mathsf{solveEqConstraints}(\{\tau_1 =_t \tau_2\}, err, \sigma)$$

where $\tau_1, \tau_2, \tau_3 \in \mathcal{T}yn\{wt\}$.

Type variables occurring in a subtyping constraint set can be indirectly limited due to the transitive property of $<:_t$. Thus, consequent constraints are computed by the application of transitive closure of $<:_t$ which also keeps the existing constraints of $\mathcal{C}$:

$$\{\tau_1 <:_t \alpha, \alpha <:_t \tau_2\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad \{\tau_1 <:_t \tau_2, \tau_1 <:_t \alpha, \alpha <:_t \tau_2\} \cup \mathcal{C}', err, \sigma$$

where $\tau_1, \tau_2 \in \mathcal{T}yn\{wt\}$ and $\alpha \in \mathcal{X}$.

The transitive closure of $<:_t$ on $\mathcal{C}$ represents the *closed form* of the subtyping constraint set that is stable via transitivity.

Definition 6.1 (Closed form)**.** *Let $\mathcal{C}$ be a subtyping constraint set. $\mathcal{C}$ is said to be* closed under transitivity*, or* closed *for short, if and only if whenever $\{\tau_1 <:_t \alpha, \alpha <:_t \tau_2\} \uplus \mathcal{C}$ such that $\tau_1, \tau_2 \in \mathcal{T}yn\{wt\}$ and $\alpha \in \mathcal{X}$, $\{\tau_1 <:_t \tau_2\}$ is defined and included in $\mathcal{C}$.*

6.2 Incompatibility detection

Since the hierarchy between types of terms is provided by the transitive closure of $<:_t$ on a context Γ, we define a rule to validate ground subtyping constraints. The objective is to detect which ground constraint of $\mathcal{C}$ are not valid with respect to the type hierarchy considered and, consequently, simplify the constraint set and possibly raise error messages.

$$\{s_1^{h_1} <:_t s_2^{h_2}\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad \mathcal{C}', \mathsf{isSub}(s_1^{h_1}, s_2^{h_2}) \cup err, \sigma$$

where $h_1, h_2 \in \mathcal{F}^\star \cup \{?\}$.

In this phase, the algorithm isSub, presented in Fig. 8, is called each time a ground subtyping constraint is found in $\mathcal{C}$. The rules for fail detection are based on the definition of the partial order $<:_t$ (see Definition 4.6) considering type hierarchy provided by Γ^*.

6.3 Canonization

This phase aims to reduce the search space of subtyping constraint resolution. The idea is to put the subtyping constraint set in *canonical form*, i.e. to limit each type variable occurring in it with at most one ground type.

Definition 6.2. *Let $\mathcal{C}$ be a subtyping constraint set. $\mathcal{C}$ is said to be in* canonical form *if and only if each type variable α has exactly one ground type as lower bound and one ground type as upper bound.*

In order to put an arbitrary subtyping constraint set in canonical form, we consider the following rules:

$$\begin{array}{lcll}
\{s_1^{h_1} <:_t \alpha, s_2^{h_2} <:_t \alpha\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \{s^? <:_t \alpha\} \cup \mathcal{C}', err, \sigma & \\
 & & \text{if } \exists s^? \text{ such that } s^? \text{ is the lub of } \{s_1^{h_1}, s_2^{h_2}\} & \\
\{s_1^{h_1} <:_t \alpha, s_2^{h_2} <:_t \alpha\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \mathcal{C}', \{(s_1^{h_1}, s_2^{h_2})\} \cup err, \sigma & \\
 & & \text{if } \nexists s^? \text{ such that } s^? \text{ is the lub of } \{s_1^{h_1}, s_2^{h_2}\} & \\
 & & & \\
\{\alpha <:_t s_1^{h_1}, \alpha <:_t s_2^{h_2}\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \{\alpha <:_t s_1^{h_1}\} \cup \mathcal{C}', err, \sigma & \\
 & & \text{if } \Gamma^*(s_1^? <:_t s_2^?) \wedge (v_1 = v_2 \vee v_2 =?) & \\
\{\alpha <:_t s_1^{h_1}, \alpha <:_t s_2^{h_2}\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \{\alpha <:_t s_2^{h_2}\} \cup \mathcal{C}', err, \sigma & \\
 & & \text{if } \Gamma^*(s_2^? <:_t s_1^?) \wedge (v_1 = v_2 \vee v_1 =?) & \\
\{\alpha <:_t s_1^{h_1}, \alpha <:_t s_2^{h_2}\} \uplus \mathcal{C}', err, \sigma & \Longrightarrow & \mathcal{C}', \{s_1^{h_1}, s_2^{h_2}\} \cup err, \sigma & \\
 & & \text{if } s_1^{h_1} \text{ and } s_2^{h_2} \text{ are not comparable} &
\end{array}$$

where $h_1, h_2 \in \mathcal{F}^\star \cup \{?\}$ and $\alpha \in \mathcal{X}$.

Errors caused by incompatible types also can be raised during the canonization process. This happens when two lower (resp. upper) bounds are not related and form a new pair (type,type) to be added to err.

6.4 Generation of solution

The subtyping constraint set $\mathcal{C}$ resulting from the sequence of simplification, closure, incompatibility detection and canonization phases (i.e. the propagation process) is said to be in *solved form* if no errors were found, that is $err = \{\}$. While solving $\mathcal{C}$ we wish to make sure, after each application of a rule for the generation of a solution, that the constraint set at hand is satisfiable, so as to detect errors as soon as possible. Therefore we combine the rules for constraint propagation with the following ones for constraint resolution:

$$\{\alpha <:_t s^h\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad [\alpha \mapsto s^h]\mathcal{C}', err, \{\alpha \mapsto s^h\} \circ \sigma$$

$$\{s^h <:_t \alpha\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad [\alpha \mapsto s^h]\mathcal{C}', err, \{\alpha \mapsto s^h\} \circ \sigma$$

where $h \in \mathcal{F}^\star \cup \{?\}$ and $\alpha \in \mathcal{X}$.

Instead of applying these rules exhaustively, only the first applicable one is applied. Thus, the order of the rules is really important as a means to opt for the least restrictive type when having two possibilities. For instance, in case a type variable α is limited by $\alpha <:_t s_1^{h_1}$ and $s_2^{h_2} <:_t \alpha$ in $\mathcal{C}$, the type $s_1^{h_1}$ will be assigned to α. As a type system for an embedded language, this approach aims to be compatible with the type system of Java in where upcasting is implicit and safe. This reduces conflicts with the type checking performed by Java that checks every cast done in the code generated by Tom.

6.5 Garbage collection

Many type variables are created during the generation of type constraints by the type inference algorithm. However, some of these type variables do not represent types of terms, but are used to indirectly limit types of input terms. In order to distinguish the two kinds of type variables, we define a set $\mathcal{I}$ of *input types*.

Definition 6.3 (Input types of a typing judgment)**.** *Consider a typing judgment $\varphi = \Gamma \vdash_{ct} e : \tau \bullet \mathcal{C}$. The set of* input types *of φ, denoted by $\mathcal{I}$, is a set that satisfies the following properties:*

- *if $e = \{rule_1; \ldots; rule_n\}$, then $\forall i \in [1, n], \forall \alpha \in \mathcal{I}, \exists \mathsf{x} \in \mathcal{V} \quad \mathsf{x} : \alpha \in \Gamma_i$*
- *if $e \neq \{rule_1; \ldots; rule_n\}$, then $\mathcal{I} = \{\}$*

Although being useful for the transitive closure computation of $\mathcal{C}$, constraints limiting only non-input type variables that do not map input type variables are no longer necessary once the propagation phase is over. Thus, they are eliminated by the following rules:

$$\{\alpha_1 <:_t \alpha_2\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad \mathcal{C}', err, \sigma \qquad \text{if } \alpha_1, \alpha_2 \notin \sigma\mathcal{I}$$

$$\{\alpha_1 <:_t \alpha_2\} \uplus \mathcal{C}', err, \sigma \quad \Longrightarrow \quad \mathcal{C}', \{\alpha_1, \alpha_2\} \cup err, \sigma \qquad \text{if } \alpha_1 \in \sigma\mathcal{I} \vee \alpha_2 \in \sigma\mathcal{I}$$

where $\alpha_1, \alpha_2 \in \mathcal{X}$.

The garbage collecting works as a verification that all input type variables are in the domain of the solution σ. This means that all variables occurring in both numeric conditions and patterns of matching conditions have their types inferred. These expressions correspond to *pure* algebraic terms and the variables occurring in it are consequently *pure* Tom variables in contrast to variables that were possibly declared in Java code.

Here comes the definition of the subtyping constraint propagation and resolution algorithm solveSubConstraints.

Definition 6.4 (Subtyping constraint resolution algorithm)**.** *The algorithm* solveSubConstraints *takes as input a 4-tuple* $(\mathcal{C}, err, \sigma, \mathcal{I})$*, where* $\mathcal{C}$ *is a subtyping constraint set,* err *is a set of type errors initially empty,* σ *is a substitution initially empty and* $\mathcal{I}$ *is a set of type variables.*

The algorithm recursively applies the rules of simplification and closure phase until a fixpoint is reached. Then it loops over $\mathcal{C}$ *until* $\mathcal{C}$ *is not modified anymore. The incompatible detection phase is then initialized in the beginning of this loop. If the resulting* err *is non-empty, then the algorithm stops and raises out all type errors found. Otherwise, the canonization phase is started.*

Afterwards, the algorithm tries to apply one of the rules of generation of solution. If at least one of them is applicable, then the first applicable one is applied to $\mathcal{C}$ *and it goes back to the beginning of the loop. Otherwise, the algorithm interrupts the loop.*

Once out of the loop, the garbage collecting is activated until $\mathcal{C}$ *becomes empty. At this moment, if* err *is empty, then the generated* σ *constitutes a solution for the original* $\mathcal{C}$*. Otherwise, all error messages collected during the resolution have been raised and the algorithm fails because either the original* $\mathcal{C}$ *has no solution or more type annotations must be provided in order to find a solution for the original* $\mathcal{C}$*.*

Note that the application of one of the generation rules keeps $\mathcal{C}$ in closed form and does not generate neither reflexive constraints nor pairs of symmetric constraints. For that reason, the simplification and closure phase are not included in the loop. The pseudocode of the algorithm is the following:

```
GET ConstraintList
SET ConstraintList to the application of the simplification and closure in it
REPEAT
    SET ConstraintList to the appplication of incompatibility detection in it
    IF Err is true THEN
        RETURN ConstraintList
    ELSE
        SET ConstraintList to the application of canonization in it
        SET ConstraintList to the application of generation of solution in it
    ENDIF
UNTIL ConstraintList has reached a fixpoint
CALL garbage collection in ConstraintList
```

Since type inference generates both an equality constraint set and a subtyping constraint set, constraint resolution is implemented by the algorithm solveConstraints which is made up of a combination of solveEqConstraints and solveSubConstraints. Initially, the equality constraint set and the empty sets err and σ are passed as arguments to solveEqConstraints. Then the returned sets err and σ in saturated form are verified. If $err \neq \emptyset$, then all errors messages collected during the equality constraint resolution have been raised and the algorithm solveConstraints fails since σ is rejected. Otherwise, σ is applied to the subtyping constraint set. The set $\mathcal{C}'$ obtained from the application of σ is, jointly with err, σ and a set $\mathcal{I}$ of input type variables, passed to solveSubConstraints. The algorithm stops when the constraint set becomes empty. If $err \neq \emptyset$, then the algorithm fails and σ is rejected. Otherwise, σ is applied to the Tom expression considered in the constraint typing judgment.

Example 6.5. *Considering Example 4.10, in the constraint typing judgment*

$$\Gamma \vdash_{ct} \{\mathsf{concNat(x*,pnat@!zero(),z*)} \prec\!|\, NatList^{?} \,|\, \mathsf{nList} \longrightarrow \text{`}\mathsf{pnat}\} : wt \bullet \mathcal{C}$$

the generated constraint set is $\mathcal{C} = \mathcal{C}_e \cup \mathcal{C}_s$*. Moreover, the generated set of input types is* $\mathcal{C} = \{\alpha_2, \alpha_3, \alpha_4\}$*.*

The equality constraint set in canonical form is $\mathcal{C}_e = \mathcal{C}_{e1} \cup \mathcal{C}_{e2}$ *where*

$\mathcal{C}_{e1} = \{\alpha_5 =_t NatList^{concNat}, \alpha_5 =_t NatList^{concNat}, \alpha_5 =_t NatList^{concNat}, \alpha_2 =_t \alpha_5\}$
$\mathcal{C}_{e2} = \{\alpha_4 =_t \alpha_7, \alpha_7 =_t ZNat^{?}, \alpha_3 =_t \alpha_5, \alpha_1 =_t \alpha_6, \alpha_4 =_t \alpha_8\}$

Let $err = \{\}$ *and* $\sigma = \{\}$*. Applying* solveEqConstraints *to* $(\mathcal{C}_e, err, \sigma)$*, we obtain* $err = \{\}$ *and* $\sigma = \{\alpha_5 \mapsto NatList^{concNat}, \alpha_2 \mapsto NatList^{concNat}, \alpha_4 \mapsto ZNat^{?}, \alpha_7 \mapsto ZNat^{?}, \alpha_3 \mapsto NatList^{concNat}, \alpha_1 \mapsto \alpha_6, \alpha_8 \mapsto ZNat^{?}\}$*.*

The subtyping constraint set is $\mathcal{C}_s = \{\alpha_5 <:_t \alpha_6, \alpha_7 <:_t Nat^{?}\}$
Let $\mathcal{C}'_s = \sigma\mathcal{C}_s = \{NatList^{concNat} <:_t \alpha_6, ZNat^{?} <:_t Nat^{?}\}$*. The application of* solveSubConstraints *to* $(\mathcal{C}'_s, err, \sigma, \mathcal{I})$ *proceeds as in the following:*

- *Simplification and closure phase returns:*
$$(\{NatList^{concNat} <:_t \alpha_6, ZNat^{?} <:_t Nat^{?}\}, err, \sigma, \mathcal{I})$$
- *Incompatibility detection phase returns:*
$$(\{NatList^{concNat} <:_t \alpha_6\}, err, \sigma, \mathcal{I})$$
- *Canonization phase returns:*
$$(\{NatList^{concNat} <:_t \alpha_6\}, err, \sigma, \mathcal{I})$$
- *Generation of solution phase returns:*
$$(\{\}, err, \{\alpha_6 \mapsto NatList^{concNat}\} \circ \sigma, \mathcal{I})$$
- *Garbage collection phase returns:*
$$(\{\}, err, \{\alpha_6 \mapsto NatList^{concNat}\} \circ \sigma, \mathcal{I})$$

Since the constraint set $\mathcal{C}$ *is empty, the algorihtm* solveConstraints *stops. Moreover,* $\{\alpha_1 \mapsto NatList^{?}, \alpha_2 \mapsto NatList^{concNat}, \alpha_3 \mapsto NatList^{concNat}, \alpha_4 \mapsto ZNat^{?}, \alpha_5 \mapsto NatList^{concNat}, \alpha_6 \mapsto NatList^{concNat}, \alpha_7 \mapsto ZNat^{?}, \alpha_8 \mapsto ZNat^{?}\}$ *is generated as a solution for* $(\Gamma, \{\mathsf{concNat(x*,pnat@!zero(),z*)} \prec\!\!\prec |NatList^{?}| \ \mathsf{nList} \longrightarrow \text{`pnat}\}, wt, \mathcal{C})$*, since* $err = \emptyset$*.*

References

[1] Emilie Balland, Claude Kirchner, and Pierre-Etienne Moreau. Formal islands. In *AMAST*, pages 51–65, 2006.

[2] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of The American Mathematical Society*, 1969.

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

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

[5] François Pottier. Type inference in the presence of subtyping: from theory to practice. Research Report 3483, INRIA, September 1998.

[6] Antoine Reilles. Canonical abstract syntax trees. *Electronic Notes in Theoretical Computer Science*, 176(4):165 – 179, 2007. Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006).

$$\frac{\mathcal{C} = \{\alpha =_t \alpha_1\}}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Var} \qquad \frac{\mathcal{C} = \{\alpha =_t \alpha_1\}}{\Gamma(\mathsf{x}^* : \alpha) \vdash_{ct} \mathsf{x}^* : \alpha_1 \bullet \mathcal{C}}\ \text{CT-SVar}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}}{\Gamma \vdash_{ct} !\mathsf{t} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Anti} \qquad \frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}_1 \quad \mathcal{C} = \{\alpha =_t \alpha_1\} \cup \mathcal{C}_1}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x}@\mathsf{t} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Alias}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_n \bullet \mathcal{C}_n \qquad \mathcal{C} = \{\alpha_0 =_t s^?\} \bigcup_{i=1}^{n} \mathcal{C}_i \cup \{\alpha_i =_t s_i^?\}}{\Gamma(\mathsf{f} : s_1^?, \ldots, s_n^? \to s^?) \vdash_{ct} \mathsf{f}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : \alpha_0 \bullet \mathcal{C}}\ \text{CT-Fun}$$

$$\frac{\mathcal{C} = \{\alpha_1 =_t s^v\}}{\Gamma(\mathsf{v} : (s_1^?)^* \to s^v) \vdash_{ct} \mathsf{v}() : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Empty}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_2 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\alpha_1 =_t s^v, \alpha_2 =_t s_1^?\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma(\mathsf{v} : (s_1^?)^* \to s^v) \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : \alpha_1 \bullet \mathcal{C}} \text{ if } \mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v \text{ and } \mathsf{t}_n \neq \mathsf{x}^*\ \text{CT-Elem}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_1 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\alpha_1 =_t s^v\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma(\mathsf{v} : (s_1^?)^* \to s^v) \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : \alpha_1 \bullet \mathcal{C}} \text{ if } \mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v \text{ or } \mathsf{t}_n = \mathsf{x}^*\ \text{CT-Merge}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}_1 \quad \mathcal{C} = \{\alpha =_t \alpha_1\} \cup \mathcal{C}_1}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x} := \text{`}\mathsf{t} : wt \bullet \mathcal{C}}\ \text{CT-Ass} \qquad \frac{\Gamma \vdash \mathsf{t} : \alpha_1 \bullet \mathcal{C}}{\Gamma \vdash \text{`}\mathsf{t} : wt \bullet \mathcal{C}}\ \text{CT-BqTerm}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_2 : \alpha_2 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\tau =_t \alpha_1, \alpha_1 =_t \alpha_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (\mathsf{t}_1 \lessdot |\tau|\ \mathsf{t}_2) : wt \bullet \mathcal{C}}\ \text{CT-Match}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_2 : \alpha_1 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (\mathsf{t}_1 \diamond \mathsf{t}_2) : wt \bullet \mathcal{C}}\ \text{CT-Num}$$

$$\frac{\Gamma \vdash_{ct} (cond_1) : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} (cond_n) : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (cond_1 \wedge \ldots \wedge cond_n) : wt \bullet \mathcal{C}}\ \text{CT-Conj}$$

$$\frac{\Gamma \vdash_{ct} (cond_1) : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} (cond_n) : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (cond_1 \vee \ldots \vee cond_n) : wt \bullet \mathcal{C}}\ \text{CT-Disj}$$

$$\frac{\Gamma \vdash_{ct} action_1 : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \vdash_{ct} action_n : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (action_1; \ldots; action_n) : wt \bullet \mathcal{C}}\ \text{CT-Action}$$

$$\frac{\Gamma \vdash_{ct} (cond) : wt \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} (action) : wt \bullet \mathcal{C}_2 \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (cond \longrightarrow action) : wt \bullet \mathcal{C}}\ \text{CT-Rule}$$

$$\frac{\Gamma \cup \Gamma_1 \vdash_{ct} rule_1 : wt \bullet \mathcal{C}_1 \quad \ldots \quad \Gamma \cup \Gamma_n \vdash_{ct} rule_n : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \bigcup_{i=1}^{n} \Gamma_i \vdash_{ct} (rule_1; \ldots; rule_n) : wt \bullet \mathcal{C}}\ \text{CT-Block}$$

where $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}$ and $\mathsf{v} \in \mathcal{F}^\star$

Figure 2: Type inference rules.

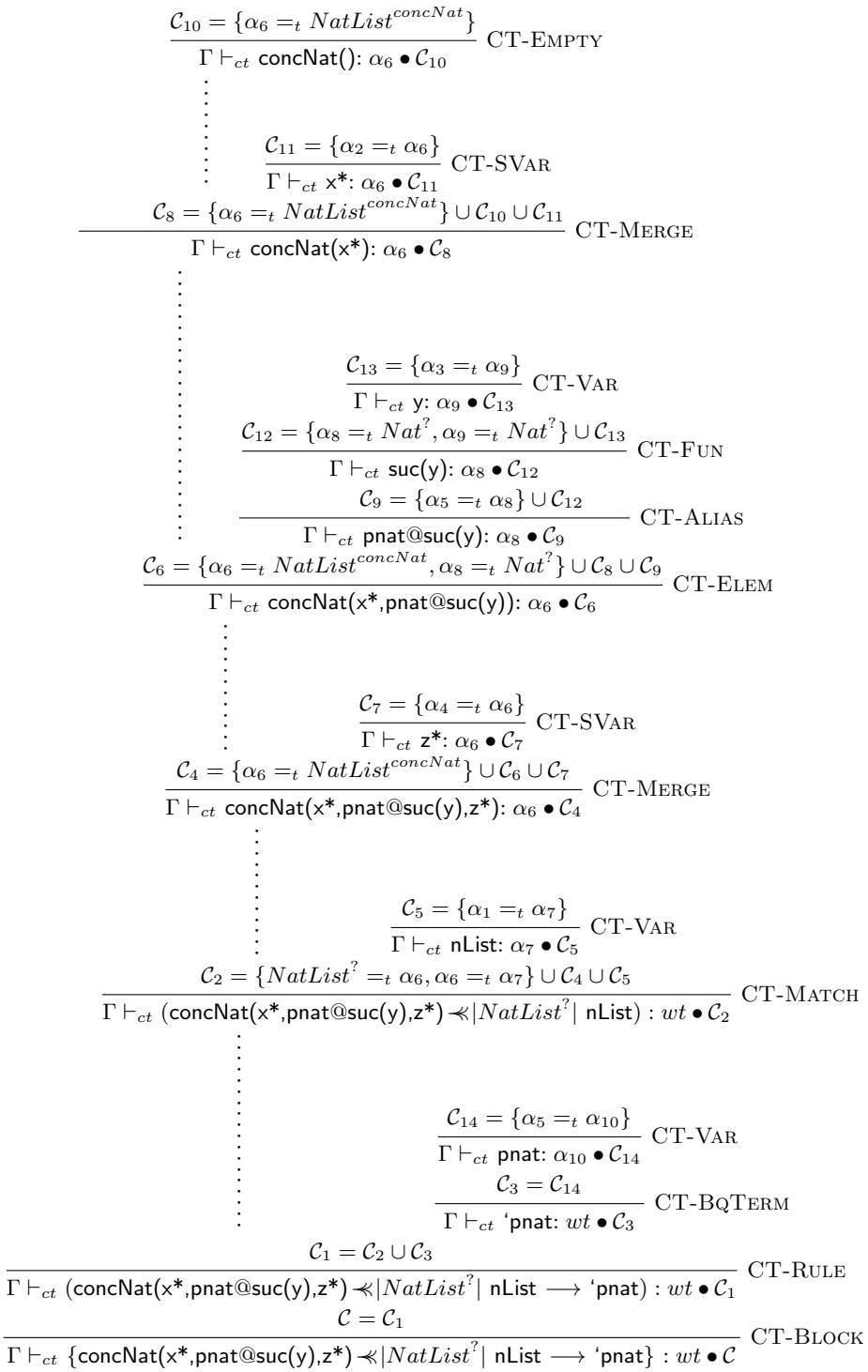

Figure 3: Example using the type inference rules.

$$\frac{\mathcal{C} = \{\alpha =_t \alpha_1\}}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Var} \qquad \frac{\mathcal{C} = \{\alpha =_t \alpha_1\}}{\Gamma(\mathsf{x}^* : \alpha) \vdash_{ct} \mathsf{x}^* : \alpha_1 \bullet \mathcal{C}}\ \text{CT-SVar}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}}{\Gamma \vdash_{ct} !\mathsf{t} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Anti} \qquad \frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}_1 \quad \mathcal{C} = \{\alpha =_t \alpha_1\} \cup \mathcal{C}_1}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x}@\mathsf{t} : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Alias}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \quad \ldots_n \quad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_n \bullet \mathcal{C}_n \qquad \mathcal{C} = \{\alpha_0 =_t s^?\} \bigcup_{i=1}^{n} \mathcal{C}_i \cup \{\alpha_i <:_t s_i^?\}}{\Gamma(\mathsf{f} : s_1^?, \ldots, s_n^? \to s^?) \vdash_{ct} \mathsf{f}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : \alpha_0 \bullet \mathcal{C}}\ \text{CT-Fun}$$

$$\frac{\mathcal{C} = \{\alpha_1 =_t s^v\}}{\Gamma(\mathsf{v} : (s_1^?)^* \to s^v) \vdash_{ct} \mathsf{v}() : \alpha_1 \bullet \mathcal{C}}\ \text{CT-Empty}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_2 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\alpha_1 =_t s^v, \alpha_2 <:_t s_1^?\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma(\mathsf{v} : (s_1^?)^* \to s^v) \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : \alpha_1 \bullet \mathcal{C}} \text{ if } \mathsf{typeof}(\Gamma,\mathsf{t}_n) \neq s^v \text{ and } \mathsf{t}_n \neq \mathsf{x}^*\ \text{CT-Elem}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_{n-1}) : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_n : \alpha_1 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\alpha_1 =_t s^v\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma(\mathsf{v} : (s_1^?)^* \to s^v) \vdash_{ct} \mathsf{v}(\mathsf{t}_1,\ldots,\mathsf{t}_n) : \alpha_1 \bullet \mathcal{C}} \text{ if } \mathsf{typeof}(\Gamma,\mathsf{t}_n) = s^v \text{ or } \mathsf{t}_n = \mathsf{x}^*\ \text{CT-Merge}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t} : \alpha_1 \bullet \mathcal{C}_1 \quad \mathcal{C} = \{\alpha =_t \alpha_1\} \cup \mathcal{C}_1}{\Gamma(\mathsf{x} : \alpha) \vdash_{ct} \mathsf{x} := \text{`}\mathsf{t} : wt \bullet \mathcal{C}}\ \text{CT-Ass} \qquad \frac{\Gamma \vdash \mathsf{t} : \alpha_1 \bullet \mathcal{C}}{\Gamma \vdash \text{`}\mathsf{t} : wt \bullet \mathcal{C}}\ \text{CT-BqTerm}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_2 : \alpha_2 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\tau =_t \alpha_1, \alpha_1 <:_t \alpha_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (\mathsf{t}_1 \lessdot|\tau|\ \mathsf{t}_2) : wt \bullet \mathcal{C}}\ \text{CT-Match}$$

$$\frac{\Gamma \vdash_{ct} \mathsf{t}_1 : \alpha_1 \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} \mathsf{t}_2 : \alpha_2 \bullet \mathcal{C}_2 \qquad \mathcal{C} = \{\alpha <:_t \alpha_1, \alpha <:_t \alpha_2\} \cup \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (\mathsf{t}_1 \diamond \mathsf{t}_2) : wt \bullet \mathcal{C}}\ \text{CT-Num}$$

$$\frac{\Gamma \vdash_{ct} (cond_1) : wt \bullet \mathcal{C}_1 \quad \ldots_n \quad \Gamma \vdash_{ct} (cond_n) : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (cond_1 \wedge \ldots \wedge cond_n) : wt \bullet \mathcal{C}}\ \text{CT-Conj}$$

$$\frac{\Gamma \vdash_{ct} (cond_1) : wt \bullet \mathcal{C}_1 \quad \ldots_n \quad \Gamma \vdash_{ct} (cond_n) : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (cond_1 \vee \ldots \vee cond_n) : wt \bullet \mathcal{C}}\ \text{CT-Disj}$$

$$\frac{\Gamma \vdash_{ct} action_1 : wt \bullet \mathcal{C}_1 \quad \ldots_n \quad \Gamma \vdash_{ct} action_n : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \vdash_{ct} (action_1; \ldots; action_n) : wt \bullet \mathcal{C}}\ \text{CT-Action}$$

$$\frac{\Gamma \vdash_{ct} (cond) : wt \bullet \mathcal{C}_1 \qquad \Gamma \vdash_{ct} (action) : wt \bullet \mathcal{C}_2 \qquad \mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2}{\Gamma \vdash_{ct} (cond \longrightarrow action) : wt \bullet \mathcal{C}}\ \text{CT-Rule}$$

$$\frac{\Gamma \cup \Gamma_1 \vdash_{ct} rule_1 : wt \bullet \mathcal{C}_1 \quad \ldots_n \quad \Gamma \cup \Gamma_n \vdash_{ct} rule_n : wt \bullet \mathcal{C}_n \qquad \mathcal{C} = \bigcup_{i=1}^{n} \mathcal{C}_i}{\Gamma \bigcup_{i=1}^{n} \Gamma_i \vdash_{ct} (rule_1; \ldots; rule_n) : wt \bullet \mathcal{C}}\ \text{CT-Block}$$

where $\mathsf{x} \in \mathcal{V}$, $\mathsf{f} \in \mathcal{F}$ and $\mathsf{v} \in \mathcal{F}^\star$

Figure 4: Type inference rules considering subtyping.

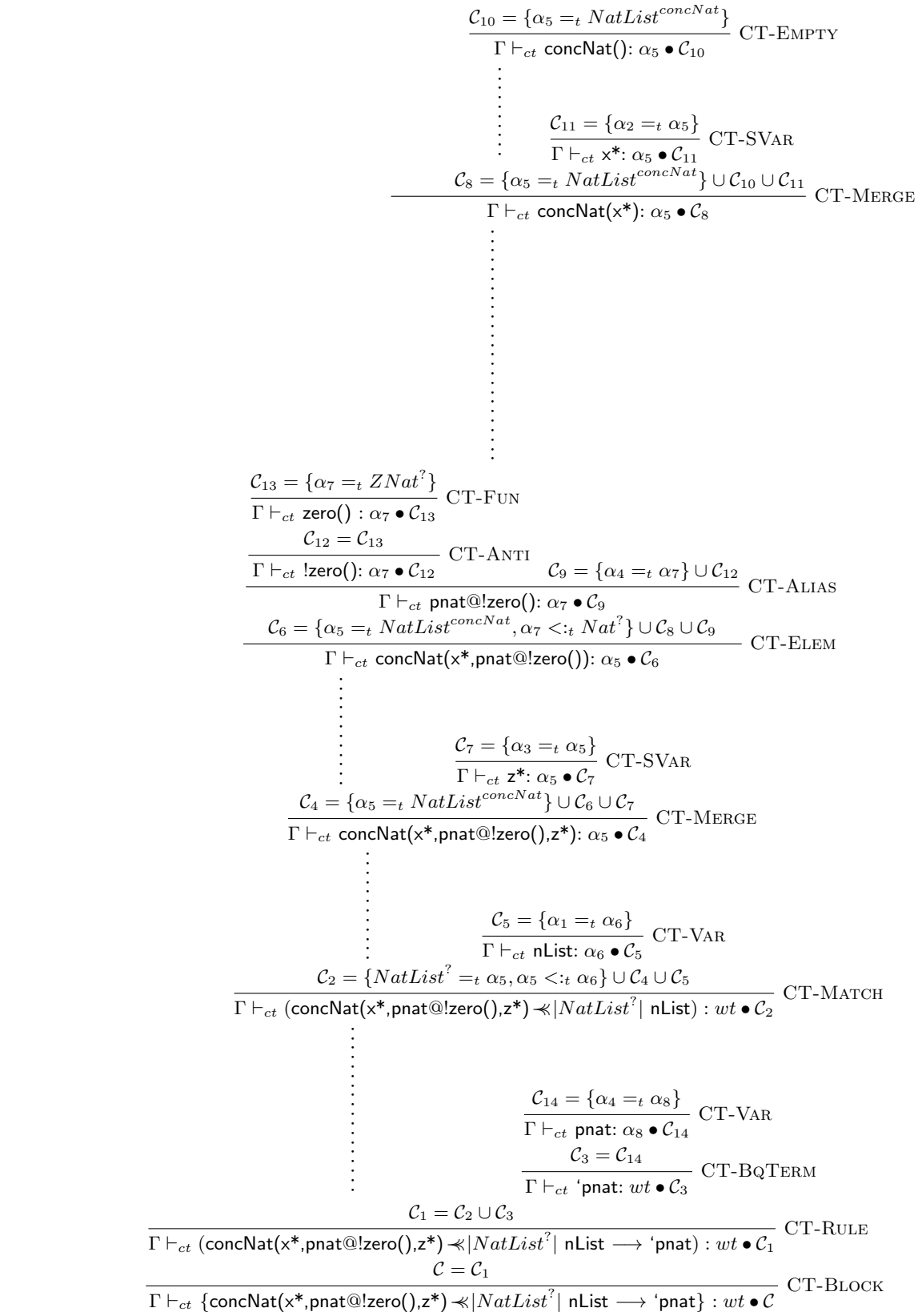

Figure 5: Example using the type inference algorithm with subtyping.

$$
\begin{array}{lllll}
(1) & \tau =_t \tau, err, \sigma & \Longrightarrow & err, \sigma & \\
\\
(2) & s_1^{h_1} =_t s_2^{h_2}, err, \sigma & \Longrightarrow & \mathsf{isEq}(s_1^{h_1}, s_2^{h_2}) \cup err,\ \sigma & \\
\\
(3a) & s_1^{h_1} =_t \alpha_1, err, \sigma & \Longrightarrow & err, \downarrow ([\alpha_1 \mapsto s_1^{h_1}] \circ \sigma) & \text{if } \sigma\alpha_1 = \alpha_1 \\
(3b) & s_1^{h_1} =_t \alpha_1, err, \sigma & \Longrightarrow & \mathsf{isEq}(s_1^{h_1}, s_2^{h_2}) \cup err, \sigma & \text{if } \sigma\alpha_1 = s_2^{h_2} \\
(3c) & s_1^{h_1} =_t \alpha_1, err, \sigma & \Longrightarrow & err, \downarrow ([\alpha_2 \mapsto s_1^{h_1}] \circ \sigma) & \text{if } \sigma\alpha_1 = \alpha_2 \\
\\
(4a) & \alpha_1 =_t \alpha_2, err, \sigma & \Longrightarrow & err, \downarrow ([\alpha_1 \mapsto \alpha_2] \circ \sigma) & \text{if } \sigma\alpha_1 = \alpha_1 \wedge \sigma\alpha_2 = \alpha_2 \\
(4b) & \alpha_1 =_t \alpha_2, err, \sigma & \Longrightarrow & err, \downarrow ([\alpha_1 \mapsto s_1^{h_1}] \circ \sigma) & \text{if } \sigma\alpha_1 = \alpha_1 \wedge \sigma\alpha_2 = s_1^{h_1} \\
(4c) & \alpha_1 =_t \alpha_2, err, \sigma & \Longrightarrow & err, \downarrow ([\alpha_1 \mapsto \alpha_3] \circ \sigma) & \text{if } \sigma\alpha_1 = \alpha_1 \wedge \sigma\alpha_2 = \alpha_3 \\
\\
(4d) & \alpha_1 =_t \alpha_2, err, \sigma & \Longrightarrow & \mathsf{isEq}(s_1^{h_1}, s_2^{h_2}) \cup err, \sigma & \text{if } \sigma\alpha_1 = s_1^{h_1} \wedge \sigma\alpha_2 = s_2^{h_2} \\
(4e) & \alpha_1 =_t \alpha_2, err, \sigma & \Longrightarrow & err, \downarrow ([\alpha_3 \mapsto s_1^{h_1}] \circ \sigma) & \text{if } \sigma\alpha_1 = s_1^{h_1} \wedge \sigma\alpha_2 = \alpha_3 \\
\\
(4f) & \alpha_1 =_t \alpha_2, err, \sigma & \Longrightarrow & err, \downarrow ([\alpha_3 \mapsto \alpha_4] \circ \sigma) & \text{if } \sigma\alpha_1 = \alpha_3 \wedge \sigma\alpha_2 = \alpha_4
\end{array}
$$

where $h_1, h_2 \in \mathcal{F}^\star \cup \{?\}$

Figure 6: The algorithm solveEqConstraints: constraint resolution rules for solving a constraint set $\mathcal{C}$.

$$
\begin{array}{lllll}
(1a) & s_1^?, s_2^h & \Longrightarrow & \{\} & \text{if } s_1 = s_2 \\
(1b) & s_1^?, s_2^h & \Longrightarrow & \{(s_1^?, s_2^h)\} & \text{if } s_1 \neq s_2 \\
\\
(2a) & s_1^h, s_2^? & \Longrightarrow & \{\} & \text{if } s_1 = s_2 \\
(2b) & s_1^h, s_2^? & \Longrightarrow & \{(s_1^h, s_2^?)\} & \text{if } s_1 \neq s_2 \\
\\
(3a) & s_1^{v_1}, s_2^{v_2} & \Longrightarrow & \{\} & \text{if } s_1 = s_2 \wedge v_1 = v_2 \\
(3b) & s_1^{v_1}, s_2^{v_2} & \Longrightarrow & \{(s_1^{v_1}, s_2^{v_2})\} & \text{if } s_1 \neq s_2 \vee v_1 \neq v_2
\end{array}
$$

where $h \in \mathcal{F}^\star \cup \{?\}$ and $v_1, v_2 \in \mathcal{F}^\star$

Figure 7: The rules of algorithm isEq for fail detection in a constraint set $\mathcal{C}$.

$$
\begin{array}{lllll}
(1a) & s_1^?, s_2^? & \Longrightarrow & \{\} & \text{if } (s_1^? <:_t s_2^?) \in \Gamma^* \\
(1b) & s_1^?, s_2^? & \Longrightarrow & \{(s_1^?, s_2^?)\} & \text{if } (s_1^? <:_t s_2^?) \notin \Gamma^* \\
\\
(2) & s_1^? <:_t s_2^v & \Longrightarrow & \{(s_1^?, s_2^v)\} & \\
\\
(3a) & s_1^v <:_t s_2^? & \Longrightarrow & \{\} & \text{if } (s_1^? <:_t s_2^?) \in \Gamma^* \\
(3b) & s_1^v <:_t s_2^? & \Longrightarrow & \{(s_1^v, s_2^?)\} & \text{if } (s_1^? <:_t s_2^?) \notin \Gamma^* \\
\\
(4a) & s_1^{v_1} <:_t s_2^{v_2} & \Longrightarrow & \{\} & \text{if } (s_1^? <:_t s_2^?) \in \Gamma^* \wedge v_1 = v_2 \\
(4b) & s_1^{v_1} <:_t s_2^{v_2} & \Longrightarrow & \{(s_1^{v_1}, s_2^{v_2})\} & \text{if } (s_1^? <:_t s_2^?) \notin \Gamma^* \vee v_1 \neq v_2
\end{array}
$$

where $v_1, v_2 \in \mathcal{F}^\star$

Figure 8: The rules of algorithm isSub for fail detection in a subtyping constraint set $\mathcal{C}$.