Type Inference for the Mixture of Matching and Implicit Subtyping

Koji Kagawa
Department of Information Science, Kagawa University, 2-1 Saiwai-cho Takamatsu, Kagawa 760, Japan
E-mail: kagawa@ec.kagawa-u.ac.jp

ABSTRACT

We present a type inference system for a language with object-oriented features such as polymorphic record access and subtyping. We separate the notion of matching and subtyping in order to avoid difficulties caused by recursive type constraints. We will use two kinds of type constraints — record (variant) constraints and subtyping constraints. Then, we discuss simplification of such mixed type constraints and show that simplification becomes much easier by the separation.

1. Introduction

In this paper, we propose an ML-style type inference system which uses constrained types and allows both polymorphic record access and implicit subtyping. We will then discuss simplification of type constraints.

In the design of type systems for object-oriented languages, much attention is paid in order to avoid difficulties caused by *binary methods* [1] — methods which take another object of the same class as an argument. The problem is that if a class has binary methods, its subclasses do not generate its subtypes. The following example is taken from [1].

$$\begin{aligned} \mathit{Point} &= \{x : \mathit{real}; y : \mathit{real}; \mathit{equal} : \mathit{MyType} \to \mathit{bool}\} \\ \mathit{ColorPoint} &= \{x : \mathit{real}; y : \mathit{real}; c : \mathit{string}; \mathit{equal} : \mathit{MyType} \to \mathit{bool}\} \end{aligned}$$

Here, *MyType* refers to the type being defined and corresponds to Eiffel's *like Current*. Both *Point* and *ColorPoint* are recursive record types and *equal* is a binary method. *ColorPoint* inherits *Point* but is not a subtype of *Point*, which can be seen from the following example:

```
procedure breakit(p: Point)
  var
    newPt: Point
  begin
    newPt := new PointClass(3.2 4.5)
    if p.equal(newPt) then
    ...
  end
```

We say τ is a *subtype* of σ if an expression of type τ can be used in any context where an expression of type σ can be used. If *ColorPoint* were a subtype of *Point* and if *cp* were an instance of *ColorPoint*, the expression *breakit(cp)* would be type-checked but would produce a runtime error.

It has been proposed recently that, in order to circumvent this difficulty, we should separate the notion of *matching* and subtyping [2]. We say that a type *match*es another if the former type has at least methods of the latter and the types of corresponding methods are the same, considering *MyType* (the type of *self*) in both types as the same. For example, if τ matches *Point*, τ has at least three methods x, y and *equal* of types $\tau \to real$, $\tau \to real$ and $\tau \to \tau \to bool$ respectively. Even if type τ matches type σ, in order for τ to be a subtype of σ, *MyType* must appear in the definition of σ only *positively*. We say a type variable appears *positively* if it appears on the left hand side of an *even* number of arrows. Similarly a type variable is said to appear *negatively* if it appears on the left hand side of an *odd* number of arrows. In the example above, *ColorPoint* matches *Point*, but is not a subtype of *Point* because *MyType* appears negatively.

Bruce [2] designed a language called PolyTOIL based on this separation. PolyTOIL supports a form of bounded polymorphism based on matching. We can write polymorphic functions which require type parameters to match another type. In PolyTOIL, type coercions are inserted automatically, while programmers must explicitly state where to use matching polymorphism. In other words, subtyping is implicit while matching is explicit. For example, a function which takes two objects matching *Point* would be written as:

```
function dist(T1<#Point; T2<#Point; a: T1; b: T2): int
```

`T1<#Point` means that a type parameter `T1` must match `Point`. Requiring such explicit declarations would be a burden to programmers, for they must "plan ahead" where to use matching polymorphism.

In practice, however, it is known by experience that matching is used much more frequently than subtyping. Therefore, it would be better to make matching implicit. What is the most desirable would be to use type inference so that programmers do not need to annotate variables with their types at all.

Type inference for languages with object-oriented features is a challenging issue. There are several proposals which extend the Damas/Milner type system, popularly used in functional languages, in order to support object-oriented features. Several extensions support *polymorphic record access* [8, 11, *etc.*]. Polymorphic record access means that we can define functions which access the same fields of more than one kind of record types. Some other extensions concern subtyping [4, 3, 10, *etc.*]. Usually, extensions of both kinds use some form of *constrained types*. A constrained type $\tau \mid C$ is a pair of two components τ and C, where τ is an ordinary type expression and C is a set of *type constraints*. For polymorphic record access, each constraint is of the

form $\alpha\#\{l:\tau\}$ which means that a type α must be a record which at least has a label l of type τ. Such constraints are generated when we use record labels. On the other hand, a subtyping constraint is of the form $\tau \triangleright \sigma$ which means that an expression of type τ is coercible to that of type σ.

In general, type inference for polymorphic record access is well studied and fairly mature now. Though existing proposals do not deal with recursive records, it seems relatively easy to add such a feature. We can consider that they offer type inference for (a limited form of) matching. They do not, however, support implicit subtyping in general. Therefore, explicit coercions must be defined and inserted by programmers when they would like to use heterogeneous collections.

On the other hand, extensions which support type inference for subtyping suffer from the fact that sets of type constraints grow too rapidly, since type constraints are usually generated for every function application. Their sizes are at least proportional to the sizes of programs. Therefore, simplification of type constraint sets becomes an essential issue. Fuh and Mishra [4] reported that by applying some simplification rules, the size of the constraint set reduces to one or zero in most programs. However, they considered only structural subtyping and did not consider subtyping between records. The Hopkins Objects Group [3] proposed a system which supports subtyping between records. It uses *recursive constraint (rc)* — i.e. constraint of the form $\alpha \triangleright \tau[\alpha]$ (or $\tau[\alpha] \triangleright \alpha$), where $\tau[\alpha]$ is a type expression which may contain α, and it can also support polymorphic record access even for recursive records. Though some simplification rules for such *rc* types were proposed, type constraints were in general too complex for humans to understand. Pottier [10] defined a more powerful entailment relation between recursive constraint sets and showed that further simplification is possible. He gave a general condition as to when we can apply substitutions to types in order to simplify type constraints and used heuristics to find such substitutions. Then the inferred types became much simpler. However, the entailment relation seems complicated, especially when it deals with recursive constraints. It seems difficult to explain the entailment relation and the simplification rule to programmers.

In this paper, we propose a type inference system which separates subtyping and matching (record polymorphism) in order to avoid recursive constraints and uses both record constraints and subtyping constraints simultaneously. They are different notions, have different implementations and therefore should be separated. Subtyping constraints are still inserted implicitly and are later simplified. For the simplification of subtyping constraints, we can use results already given in the previous work. We will then present some cases where simplification of mixed type constraint sets is possible.

We require that recursive record (and variant) types should be explicitly declared. Since subtyping constraints are no longer used in order to unfold recursive types, the complexity of simplification is considerably reduced.

The rest of the paper is organized as follows. Section 2 gives an overview of our system. Section 3 explains the simplification rules of constraint sets. Section 4 presents some examples. Section 5 concludes.

2. Overview

The expression language is an ordinary λ-calculus with polymorphic `let`.

$$e ::= x \parallel \lambda x.e \parallel e\ e \parallel \texttt{let}\ x = e\ \texttt{in}\ e$$

Though the full language would have constructs for building and accessing pairs and records (and variants), they are all viewed as primitive functions.

The type language is given as follows:

Types			
	τ	$::= \alpha$	type variables
		$\parallel \tau \to \tau$	function types
		$\parallel \ldots$	other constructed types
Type constraints			
	c	$::= \tau \# \tau$	matching (record) constraints
		$\parallel \tau \triangleright \tau$	subtyping constraints
Type schemes			
	κ	$::= \forall \overline{\alpha}.\tau \mid \{c_i\}$	

"Other constructed types" above includes record (and variant) types and they can be recursive. Like most existing systems for subtyping and record polymorphism, we will use constrained types — pairs of an ordinary type expression (referred to as a *type body*), and a constraint set. We write a constrained type as $\tau \mid C$ where C is the constraint set and τ is the type body. We use both *matching (record) constraints* of the form $\tau \# \tau$ and *subtyping constraints* of the form $\tau \triangleright \tau$ in a single set. $\tau \# \sigma$ reads "τ *matches* σ" and it indicates that a type τ must have labels of σ.

For simplicity, we require explicit declaration of record labels so that we can use *recursive* records[a]. Though this may seem awkward, it simplifies the type inference process and would be worth the costs. In most functional languages, we are already required to declare recursive datatypes in advance. For example, in ML and Haskell, this is done by `datatype` and `data` declarations respectively. Therefore, we can expect our requirement here would not cause so much inconvenience for programmers.

In this paper, we will use the following syntax:

record $Null = \{\ null : bool\ \}$
record $Head(\alpha) = \{\ head : \alpha\ \}$

[a] Extension to polymorphic variants would be discussed later.

By these declarations, two labels *null* and *head* become available. For convenience, we give a name to each set of record constraints. *Null* and *Head* above are such names. They are used as shorthands for sets of record constraints.

Then *head*[b] has type $\gamma \to \alpha \,|\, \{\gamma \# Head(\alpha)\}$ and can be used as a selector function for records which have label *head*. Similarly, *null* has type $\gamma \to bool \,|\, \{\gamma \# Null\}$. The type constraint $\{\gamma \# Head(\alpha)\}$ means that γ is a record type and has a label *head* of type α. Such type constraints are added to the constraint set when we use selector functions such as *head*. Recursive labels are declared using the keyword *MyType*. For example,

record *Tail* = { *tail* : *MyType* }

declares the interface for list-like structures. Then, *tail* has type $\gamma \to \gamma \,|\, \{\gamma \# Tail\}$. For convenience, we also give a name to the constraint set which contains all the three labels above.

record *Stream*(α) **inherits** *Null, Head*(α), *Tail*

Then a type constraint $\gamma \# Stream(\alpha)$ means that γ is a record type which has at least three labels *null, head* and *tail* with types *bool*, α and γ respectively.

As can be seen from the examples above, record constraints, in general, have type parameters (e.g. α in $\gamma \# Head(\alpha)$). Note that when a type has type constraints of the same label with different parameters, the corresponding parameters must be unified. For example, if γ is constrained as $\{\gamma \# Head(\alpha), \gamma \# Head(\beta), \ldots\}$ then two parameters α and β must be unified. This is a natural requirement, since a field in a single record type cannot have more than one type. We will refer to parameters before # (e.g. γ above) as *independent* parameters and parameters after # (e.g. α and β above) as *dependent* parameters.

Strictly speaking, we should distinguish *interfaces* (constraints) and *types*. An interface is a set of labels paired with their types. Matching is a relation between a type and an interface, not between two types. Of course, each interface has its *standard* type — a record type which exactly has labels specified in the interface. For example, the standard type of a constraint set *Stream*(α) is a recursive record type:

$$\textbf{type } Stream(\alpha) \;\; = \;\; \mu x.\{head : \alpha, tail : x, null : bool\}$$

where $\mu x.\tau[x]$ stands for the fixed point of $x \mapsto \tau[x]$. Therefore, we will use a name for a constraint also as a name for its standard type. Then, record expressions such as:

empty = { *null* = *True, head* = *error, tail* = *error* }
ex1 = { *null* = *False, head* = *1, tail* = *empty* }

[b] In this paper, we do not introduce a special syntax in order to access record fields. Selector functions are ordinary functions except that they have constrained types. We do not consider operators which overwrite fields, though it is straightforward to add such operators.

have standard types ($empty : Stream(\alpha)$ and $ex1 : Stream(int)$), and the expression $head\ ex1$ evaluates to 1. What is important is that we can define coercions into the standard type from any types that match the interface, provided that $MyType$ appears only positively in the interface. Such coercion functions are the "packing" functions of Hofmann and Pierce [5], which are defined using "map" functions:

$$map_N : (x \to y) \to (N(x) \to N(y))$$

associated with each positive type operator N.

The type inference rules are given in Figure 1. Type judgments are of the form $A \vdash e : \tau \mid C$ where A is a type environment. Type coercions are inserted automati-

$$\frac{A(x) = \forall \overline{\alpha}.\tau \mid C \quad \varphi \text{ is a substitution of domain } \overline{\alpha}}{A \vdash x : \varphi(\tau \mid C)}(\text{Var})$$

$$\frac{A; x : \tau \vdash e : \tau' \mid C}{A \vdash \lambda x.e : \tau \to \tau' \mid C}(\text{Lambda})$$

$$\frac{A \vdash e_1 : \tau_1 \to \tau \mid C_1 \quad A \vdash e_2 : \tau_2 \mid C_2}{A \vdash e_1\ e_2 : \tau \mid C_1 \cup C_2 \cup \{\tau_2 \triangleright \tau_1\}}(\text{App})$$

$$\frac{\begin{array}{c} A \vdash e_1 : \tau_1 \mid C_1 \quad A; x : \forall \overline{\alpha}.\tau_1 \mid C_1 \vdash e_2 : \tau_2 \mid C_2 \\ \overline{\alpha} = \mathrm{FV}(\tau_1 \mid C_1) \setminus \mathrm{FV}(A) \quad \varphi \text{ is a substitution of domain } \overline{\alpha} \end{array}}{A \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \mid \varphi(C_1) \cup C_2}(\text{Let})$$

$$\frac{A \vdash e : \tau \mid C \quad C' \Vdash C \quad C' \Vdash \tau \triangleright \tau'}{A \vdash e : \tau' \mid C'}(\text{Subtype})$$

Figure 1: Type inference rules

cally by (App). We use type constraints of the form $\tau \triangleright \sigma$ in order to show that τ is a subtype of σ (an expression of type τ is coercible to type σ). They are introduced when we apply a function of type $\tau_1 \to \tau$ to an argument of type τ_2 by (App). Note that this rule is different form the corresponding rule in [10]:

$$\frac{A \vdash e_1 : \tau_1 \mid C_1 \quad A \vdash e_2 : \tau_2 \mid C_2}{A \vdash e_1\ e_2 : \tau \mid C_1 \cup C_2 \cup \{\tau_1 \triangleright \tau_2 \to \tau\}}(\text{App}')$$

This is because we can take the "greedy" approach — τ_1 in (App$'$) is replaced with $\tau_3 \to \tau_4$ and the constraint will be broken into more elementary ones. The result is equivalent to the former rule. Since we do not need to deal with recursive subtyping

constraints, we do not have to care about termination of such greedy decomposition of constraints.

A set of type constraints would grow rapidly during type inference. In the next section, we will explain how such constraints are *simplified*. Simplification would take place, in typical implementation, when we generalize types for `let`-bind identifiers.

In our framework, selector functions introduce record constraints and subtyping constraints are inserted with function applications. Other aspects of type inference are almost the same as existing systems for constrained types. Jones [6] proposed a very general framework for simplification of such constrained (*qualified*, in his terminology) types.

3. Simplifying type constraints

In this section, we will explain how type constraints are simplified which are generated during type inference.

3.1. Entailment

After assembling type constraints, we calculate the *closure* of subtyping constraints. A constraint set C is *closed* if and only if the following conditions hold.

$$\begin{array}{lrcl}
\text{(trans)} & \tau_1 \triangleright \alpha \in C \wedge \alpha \triangleright \tau_3 \in C & \Rightarrow & \tau_1 \triangleright \tau_3 \in C \\
\text{(arrow)} & \sigma_1 \to \sigma_2 \triangleright \tau_1 \to \tau_2 \in C & \Rightarrow & \tau_1 \triangleright \sigma_1 \in C \wedge \sigma_2 \triangleright \tau_2 \in C \\
\text{(struct)} & t(\overline{\sigma}) \triangleright t(\overline{\tau}) \in C & \Rightarrow & \tau_i ?_{t,i} \sigma_i \in C
\end{array}$$

($\overline{\tau}$ stands for a sequence of parameters $\tau_1, \ldots, \tau_n$. $?_{t,i}$ is $\triangleright$ when the ith parameter of t appears only positively in the definition of t, $\triangleleft$ when it appears only negatively, and $=$ when it appears both positively and negatively.) C^∞, the *closure* of C, is the smallest closed set containing C.

When a parameter of a subtyping constraint is neither a type variable nor a record type, we reduce it into subtyping relations between its parameters. For example, $(\alpha_1 \to \beta_1) \triangleright (\alpha_2 \to \beta_2)$ is decomposed into $\alpha_2 \triangleright \alpha_1$, $\beta_1 \triangleright \beta_2$. Moreover, $(\alpha \to \beta) \triangleright \gamma$ would be reduced into $\alpha' \triangleright \alpha$, $\beta \triangleright \beta'$ substituting γ with $\alpha' \to \beta'$. We do not need to take the "lazy" approach as in [10] since subtyping constraints are never recursive.

When the supertype parameter of a subtyping relation is a record type ($\gamma \triangleright r(\overline{\tau})$ — here and in the following, we use the following convention: r is a record type constructor or a record constraint, v is a variant type constructor or a variant constraint, and t can be used as any type constructor or any constraint.), it will be decomposed into record constraints and subtyping relations between parameters. For example,$\gamma \triangleright$ $\mathit{Stream}(\alpha)$ would be decomposed into $\gamma \# \mathit{Stream}(\beta)$ $(= \gamma \# \mathit{Null}, \gamma \# \mathit{Tail}, \gamma \# \mathit{Head}(\beta))$ and $\beta \triangleright \alpha$. That is, in order for γ to be a subtype of $\mathit{Stream}(\alpha)$, γ must also match the interface corresponding to $\mathit{Stream}(\beta)$ and β must be a subtype of α.

More generally, if we have two constraints $\tau \triangleright \alpha$ and $\alpha \# r(\overline{\sigma})$ in C, the record constraint propagate back as $\tau \# r(\overline{\beta})$ and $\beta_i ?_{r,i} \sigma_i$. Note however that, if *MyType* has a negative occurrence in the definition of r, there is no subtype of α and α must be unified with τ. In the following, we will assume records do not have binary methods. To sum things up, we add the following rule to compute the *closure* of a constraint set as well as (trans), (arrow) and (struct):

$$\begin{array}{lrcl} \text{(record1)} & \gamma \triangleright r(\overline{\sigma}) \in C & \Rightarrow & \gamma \# r(\overline{\alpha}) \in C \wedge \alpha_i ?_{r,i} \sigma_i \in C \\ \text{(record2)} & \tau \triangleright \alpha \in C \wedge \alpha \# r(\overline{\sigma}) \in C & \Rightarrow & \tau \# r(\overline{\beta}) \in C \wedge \beta_i ?_{r,i} \sigma_i \in C \end{array}$$

In the opposite case, that is, when the subtype parameter of a subtyping constraint is a record type but the supertype is a variable $(r(\overline{\sigma}) \triangleright \gamma)$, the constraint is left as it is. That is, record constraints do *not* propagate *forward.*

We check then that constraints generated by structural propagation and by transitive closures do not contain inconsistent constraints such as $\alpha * \beta \triangleright \gamma \rightarrow \delta$ and $(\alpha \rightarrow \beta) \# \mathit{Stream}(\gamma)$.

We must also do "occur check" here so that subtyping of the form $\alpha \# \tau[\alpha]$, $\alpha \triangleright \tau[\alpha]$ or $\tau[\alpha] \triangleright \alpha$ are not used, where $\tau[\alpha]$ is some type expression containing α. For example, a type constraint, $\alpha \triangleright (\alpha \rightarrow \beta)$ is forbidden. Then, type variables are stratified into several layers so that a type variable which belongs to level 0 is not used as type parameters of any other variables in the constraint set and that a variable which belongs to level i is not used as parameters of variables of level $j (\geq i)$. (We say a type variable α is used as a parameter of a type variable γ, if one of the following form $(\gamma \# r(\ldots, \alpha, \ldots)$ and $r(\ldots, \alpha, \ldots) \triangleright \gamma)$ is found in the constraint set.)

At this stage, remaining atomic constraints are either record constraints or subtyping constraints of the form $\alpha \triangleright \beta$ and $r(\overline{\tau}) \triangleright \beta$where α and β are type variables. Then we can think of a set of atomic subtyping relations as a graph of type variables where a directed edge between two variables indicate that the source node is a subtype of the destination node. *External (observable)* variables are variables which appear in the type body. Other variables are said to be *internal.*

The entailment relation is defined in Figure 2. $C \Vdash C'$ reads "C *entails* C'. The (RECORD) rule states that a type that matches a record constraint can be coerced into its standard type.

3.2. Simplification

For simplification of subtyping constraints, we can use the substitution rule in [9]:

$$\text{(Subst)} \quad \frac{A \vdash e : \tau \mid C \quad \Gamma(A) = A \quad C \Vdash \Gamma(C) \quad C \Vdash \Gamma(\tau) \triangleright \tau}{A \vdash e : \Gamma(\tau \mid C)}$$

where Γ is a substitution. The (Subst) rule means that if the effect of a substitution can be canceled later by (Subtype), we can apply the substitution in order to simplify

$$\text{(AXIOM)}\quad \frac{\tau_1 \triangleright \tau_2 \in C^\infty}{C \Vdash \{\tau_1 \triangleright \tau_2\}} \qquad \text{(REFLEX)}\quad C \Vdash \{\tau \triangleright \tau\}$$

$$\text{(TRANS)}\quad \frac{C \Vdash \{\tau_1 \triangleright \tau_2, \tau_2 \triangleright \tau_3\}}{C \Vdash \{\tau_1 \triangleright \tau_3\}} \qquad \text{(ARROW)}\quad \frac{C \Vdash \{\tau_1 \triangleright \sigma_1, \sigma_2 \triangleright \tau_2\}}{\sigma_1 \to \sigma_2 \triangleright \tau_1 \to \tau_2}$$

$$\text{(STRUCT)}\quad \frac{C \Vdash \{\sigma_i ?_{t,i} \tau_i\}}{C \Vdash \{t(\overline{\sigma}) \triangleright t(\overline{\tau})\}} \qquad \text{(RECORD)}\quad \frac{C \Vdash \{\alpha \# r(\overline{\sigma}), \sigma_i ?_{r,i} \tau_i\}}{C \Vdash \{\alpha \triangleright r(\overline{\tau})\}}$$

$$\text{(SET)}\quad \frac{\forall c \in C'.\ C \Vdash \{c\}}{C \Vdash C'}$$

Figure 2: Entailment relation

the type "without fear of failure in the future." Typically, this rule is used after type checking the body of `let`-bound variables.

Instances of this general rule which are especially useful are the following.

- Cycles of type variables can be eliminated by identifying them.
- If a type variable appears only positively (resp. negatively) in the type body and has a unique lower (resp. upper), it can be substituted by the lower (resp. upper) bound.

Actually, these rules have been already used in [4]. Note that even if a dependent parameter of some record (or variant) constraint does not appear in the body, it must be treated as appearing free if its independent parameter appears in the body. Free variables (FV) must be defined taking this into account. The definition of positive and negative appearances for such dependent parameters can be also given similarly.

Another instance of the (Subst) rule which is specific to our system for mixed constraint sets is,

- If a type variables appears only negatively in the type body and if there are some record constraints but no outgoing edges, we can substitute the variable with the standard type of the record constraints.

$$\text{(RecSubst)}\quad \frac{\begin{array}{c} A \vdash e : \tau \mid C \quad \gamma \# r(\overline{\sigma}) \in C \quad \gamma \notin FV(A) \quad \nexists \delta.\gamma \triangleright \delta \in C \\ \gamma \text{ appears only negatively in } \tau \end{array}}{A \vdash e : (\tau \mid C)[r(\overline{\sigma})/\gamma]}$$

where $[\tau/\alpha]$ stands for the substitution of α with τ.

In other words, if there is no possibility of further constraining γ, γ can be assigned the standard type of the given record constraints. This rule is also applicable when $\gamma \notin FV(\tau)$. Note that when γ appears negatively in τ, we can apply the (Subtype) rule and can "internalize" γ. Therefore, we can think that essentially, the (RecSubst) rule is used only when γ is internal.

3.3. About coherence

When we substitute internal variables, we must make sure that the result of the expression does not depend on the substituted type. This property is referred to as *coherence* [7]. We will explain the case for the (RecSubst) rule in the previous subsection.

We will briefly explain by using *parametricity*, or so-called the "free theorem" [13]. However, since we have not defined translation or "compilation" of our calculus into a more elementary one without constrained types, the explanation below is rather informal.

Suppose that $A \vdash e : \tau \mid C$, $\gamma \# r(\overline{\sigma}) \in C$, $\gamma \notin FV(\tau) \cup FV(A)$, and let $\{t_i \triangleright \gamma\}_{i \in \{1,\ldots,m\}}$ be all the incoming edges of γ and $r(\overline{\sigma})$ be a record constraint $\{l_j : \tau_j\}_j$ Let r_1 be the standard type of the constraint $r(\overline{\sigma})$ and r_2 be another record type which also satisfies the constraint for γ. The point is that we can think of e, which has a constrained type $\tau \mid C$, as a polymorphic function of type $\forall\gamma.\{t_i \to \gamma\}_{i \in \{1,\ldots,m\}} \to \{\gamma \to \tau_j[\gamma/MyType]\}_{j \in \{1,\ldots,n\}} \to \tau$ by considering coercion functions $(\{inj_{t_i \triangleright \gamma}\}_i)$ and selector functions $(\{l_j^\gamma\}_j)$ as hidden parameters. Then the free theorem implies that if the corresponding arguments are related, the results are also related.

We would like to show that $[r_1/\gamma]$ and $[r_2/\gamma]$ will give the same value to e. Note that there is a type coercion $inj_{r_2 \triangleright r_1}$ between r_2 and r_1. Let $inj_{t_i \triangleright r_k}$ ($k = 1$ or 2) be a coercion between t_i and r_k, let $l_j^{r_k}$ ($j \in \{1, \ldots, n\}$) be a selector function for r_k of type $r_k \to \tau_j[r_k/MyType]$ and let map_j be a function of type $(x \to y) \to \tau_j[x/MyType] \to \tau_j[y/MyType]$ which can be defined inductively when $MyType$ appears only positively in τ_j. Then the following equations hold if coercions $(inj_)$ are defined as in [5].

$$\begin{aligned} inj_{t_i \triangleright r_1} &= inj_{r_2 \triangleright r_1} \circ inj_{t_i \triangleright r_2} \\ l_j^{r_1} \circ inj_{r_2 \triangleright r_1} &= map_j\, inj_{r_2 \triangleright r_1} \circ l_j^{r_2} \end{aligned}$$

The equations above state that coercion functions and selector functions for r_1 and r_2 are all related by $inj_{r_2 \triangleright r_1}$. When applying substitutions $[r_1/\gamma]$ and $[r_2/\gamma]$, they are supplied as actual arguments for the "hidden parameters." From $\gamma \notin FV(\tau)$, it implies that the results are in fact the same whether the substituted type is r_1 or r_2.

3.4. Polymorphic variants

Variants are dual constructs of records. We can easily add polymorphic variants as well. Like records, we require that we should explicitly declare variant tags when we deal with recursive variant types.

We define polymorphic variant types by the following syntax:

variant $Nil = [nil]$
variant $Cons(\alpha) = [cons\ of\ \alpha * MyType]$

variant $List(\alpha)$ **inherits** $Nil,\ Cons(\alpha)$

Then two "constructor" functions are defined — *nil* has type $\gamma \mid \{\gamma \# Nil\}$ and *cons* has type $\alpha * \gamma \to \gamma \mid \{\gamma \# Cons(\alpha)\}$. They are used as "tags." Analogously to records, we can define the *standard* type for variant constraints. The standard type of $List(\alpha)$ above is the type of ordinary lists.

For records, explicit record expressions have standard record types, while for variants, pattern matching requires its argument to be of a standard variant type[c].

Here, we will only briefly show the rules for polymorphic variants, which are dual to the rules for records. The first rule is used to calculate closures of constraint sets. The second rule is added to the definition of entailment. The last one is another instance of the substitution rule.

- (Closure)

$$\begin{array}{lrcl} \text{(variant1)} & v(\overline{\sigma}) \triangleright \gamma \in C & \Rightarrow & \gamma \# v(\overline{\beta}) \in C \wedge \sigma_i ?_{v,i} \beta_i \in C \\ \text{(variant2)} & \alpha \triangleright \tau \in C \wedge \alpha \# v(\overline{\sigma}) \in C & \Rightarrow & \tau \# v(\overline{\beta}) \in C \wedge \sigma_i ?_{r,i} \beta_i \in C \end{array}$$

  where $?_{v,i}$ is $\triangleright$ when the ith parameter of v appears only positively, $\triangleleft$ when it appears only negatively, and $=$ when it appears both positively and negatively.

- (Entailment Relation)

$$\text{(VARIANT)} \quad \frac{C \Vdash \{\alpha \# v(\overline{\tau}), \sigma_i ?_{r,i} \tau_i\}}{C \Vdash \{v(\overline{\sigma}) \triangleright \alpha\}}$$

- (Substitution Rule)
  If a type variables appears only positively in the type body and if there are some variant constraints but no incoming edges, we can substitute the variable with the standard type of the variant constraints.

$$\text{(VarSubst)} \quad \frac{\begin{array}{c} A \vdash e : \tau \mid C \quad \gamma \# v(\overline{\sigma}) \in C \quad \gamma \notin FV(A) \quad \nexists \delta . \delta \triangleright \gamma \in C \\ \gamma \text{ appears only positively in } \tau \end{array}}{A \vdash e : (\tau \mid C)[v(\overline{\sigma})/\alpha]}$$

[c]We do not consider variable patterns which match any tags.

4. Examples

In this section, we show some examples. For the sake of comparison, we will use the same examples as [9]. However, since our experimental implementation is not fully operational so far, part of type inference is done by hand.

First, `map` function defined as:

```
map f nil  = nil
map f (cons (x, xs)) = cons (f x, map f xs)
```

has type $(\alpha \to \beta) \to List(\gamma) \to \delta$ with an unsimplified constraint set as follows:

$$\{\delta \# List\ \epsilon, \phi \triangleright \delta, \phi \# List\ \psi, (List\ \gamma) \triangleright (List\ \rho), (\alpha \to \beta) \triangleright (\rho \to \psi), \beta \triangleright \epsilon, \gamma \triangleright \alpha\}$$

However, by applying the substitutions $([\alpha/\rho], [\alpha/\gamma], [List(\psi)/\rho], [\epsilon/\psi], [\beta/\epsilon], [List(\beta)/\delta])$ in this order, this type is simplified to:

$$(\alpha \to \beta) \to List(\alpha) \to List(\beta)$$

After simplification, there is no type constraint left.

The *quicksort* algorithm:

```
concat nil            ys        = ys
concat (cons (x, xs)) ys        = cons (x, concat xs ys)

split comp pivot nil            = (nil,nil)
split comp pivot (cons (x, xs)) = let (ss,gs) = split comp pivot xs
                                  in if comp x pivot
                                     then (cons (x, ss),gs)
                                     else (ss, cons (x, gs))

qsort comp nil                  = nil
qsort comp (cons (x, xs))       = let (ss,gs) = split comp x xs
                                  in concat (qsort comp ss)
                                            (cons (x, qsort comp gs))
```

will produce unsimplified constraint sets of fairly large size. However, after simplification, their types become:

$$\begin{aligned} concat &: List(\alpha) \to \gamma \to \gamma \mid \{\gamma \# List(\alpha)\} \\ split &: (\alpha \to \beta \to bool) \to \beta \to List(\alpha) \to List(\alpha) * List(\alpha) \\ qsort &: (\alpha \to \alpha \to bool) \to List(\alpha) \to List(\alpha) \end{aligned}$$

Note that *concat* has a type a little more general than the regular ML type. During the simplification, we only used the instances of the substitution rule shown in the previous section.

5. Conclusion

We studied type inference for an ML-like calculus with implicit subtyping and polymorphic records and variants.

Recently, Reppy and Riecke [12] proposed a type system which supports type inference for implicit subtyping. It is based on an approach quite different from ours. It does not show constrained types to programmers. As a result it sometimes lose type information and we must use "type case" construct to recover type information. It would be interesting to study which approach is easier for programmers to understand.

Based on constrained types, we proposed separating the notion of matching and subtype in type inference. By replacing recursive constraints by matching, we expect that in most practical programs, the size of the constraint set is zero or one as Fuh and Mishra reported in [4]. A couple of examples support this observation. Compared with Pottier's work [9, 10], our system allows obviously less programs to be type-checked. For example, Pottier's system allows the (call-by-value) Y operator to be typed while ours does not. Ours supports only subtyping between records (and between variants). It would be debatable whether it is desirable to allow the Y operator to be typed.

It is worth noticing that if automatic insertion of subtyping relations and their simplification should turn out to be inefficient, or if programmers dislike such automatic insertion for some reason, it is still possible to adopt some semi-automatic approach. The type checker can quit inserting subtyping constraints automatically and may require programmers to specify where to insert a coercion — without requiring how coercions are actually defined and which types are actually used. Or we can restrict the places where coercions are automatically inserted — constructor functions and assignments are good candidates. Then, the simplification process would be much simpler but we can expect that it would cover most cases where we need subtyping and would be still practical enough.

Nevertheless, we expect that type inference with implicit subtyping is efficient enough. The implementation of the prototype type-checker is behind schedule and we cannot show the time to compute and simplify types here. Therefore, efficiency consideration as well as formalism are left as future work.

References

[1] Kim Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Ben-

jamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[2] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In W. Olthoff, editor, *Proceedings ECOOP'95 (LNCS 952)*, pages 27–51. Springer-Verlag, August 1995. Aarhus, Denmark.

[3] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proc. of the Tenth Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA95)*, pages 169–184, October 1995. Austin, Texas, USA.

[4] You Chin Fuh and Prateek Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In J. Díaz F. Orejas, editor, *TAPSOFT'89 Proceedings of the International Joint Conference on Theory and Practice of Software Development*, pages 167–183. Springer-Verlag, March 1989. Lecture Notes in Computer Science 352.

[5] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, October 1995.

[6] Mark P. Jones. Simplifying and improving qualified types. Reserch Report YALEU/DCS/RR-1040, Yale University, June 1994.

[7] M.P. Jones. Coherence for qualified types. Research Report YALEU/DCS/RR-989, Yale University, New Haven, Connecticut, USA, September 1993.

[8] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *Annual ACM Symp. on Principles of Prog. Languages*, pages 154–165, January 1992.

[9] François Pottier. Type inference and simplification for recursively constrained types. In *Actes du GDR Programmation 1995 (journée du pôle Programmation Fonctionnelle)*, November 1995.

[10] François Pottier. Simplifying subtyping constraints. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, pages 122–133, January 1996.

[11] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *TACS '94: Conference on theoretical aspects of computer software(LNCS 789)*. Springer-Verlag, April 1994. Sendai, Japan.

[12] John H. Reppy and Jon G. Riecke. Simple objects for Standard ML. In *1996 Conference on Programming Language Design and Implementation*, pages 171–180. ACM Press, 1996.

[13] Philip Wadler. Theorems for free! In *Proc. ACM Conf. Functional Programming and Computer Architecture*, pages 347–359, 1989.