数理解析研究所講究録
1023 巻 1998 年 142-155

Simplification of Subtyping Constraints and Its Application for Monadic Programming

Koji KAGAWA
Dept. of Reliability-based Information Systems Engineering
Faculty of Engineering
Kagawa University
1-1 Saïwai-cho, Takamatsu 760-0016, JAPAN
kagawa@eng.kagawa-u.ac.jp

December 1, 1997

Abstract

We discuss type inference for a language which supports both polymorphic records (variants) and implicit subtyping — useful features for object-oriented programming. We extend subtyping relation to type constructor variables. We show that such extension is especially useful for typing monadic-style functional programs, that is, imperative-style programs written in a purely functional language such as Haskell. It gives reasonably simple types to monadic (imperative) programs and is, in a sense, similar to the effect system.

We use the notion of constrained types. The point is to separate *matching* constraints and *subtyping* constraints in order to avoid the difficulties caused by recursive type constraints. We will focus on simplification of such mixed constraints.

1 Introduction

Monadic style is now becoming more and more popular in purely functional programming community. It uses two fundamental operators overloaded using a constructor class of Haskell [3]:

```
class Monad m where
    return :: a -> m a
    (>>=)  :: m b -> (b -> m a) -> m a
```

It enables us to write programs with various imperative features in purely functional languages. For example, we can write something like:

```
readVar x          >>= λ x' →
writeVar x (x'+1) >>= λ _  →
writeVar a x'
```

to increment a reference. However, monadic style functional programs are still a bit awkward, since we must introduce variables for all the intermediate results such as `x'` above. A well-known criticism for this is that monadic style resembles assembly programs. Can we write imperative programs in a more natural notation — like `a = x++`? One way to do this is to use the following operator

```
(@@) :: m # Monad => m (a -> m b) -> m a -> m b
k @@ x = k >>= λ f → x >>= f
```

in place of usual function applications (juxtaposition) like `f @@ a @@ b`. A more radical way would be to overload the notation of the usual function applications and translate expressions as follows [13]:

$$\frac{A(x) = \tau}{A \vdash' x \rightsquigarrow \mathit{return}\, x :: m\, \tau}\ (\text{M-Var})$$

$$\frac{A; x :: \tau \vdash e \rightsquigarrow e^* :: m\, \tau'}{A; \vdash \lambda x.e \rightsquigarrow \mathit{return}\, (\lambda x.e^*) :: m\, (\tau \rightarrow m\, \tau')}\ (\text{M-Lambda})$$

$$\frac{A \vdash e_1 \rightsquigarrow e_1^* :: m\, (\tau_2 \rightarrow m\, \tau_1) \quad A \vdash e_2 \rightsquigarrow e_2^* :: m\, \tau_2}{A \vdash e_1\, e_2 \rightsquigarrow e_1^*\, `\mathit{bind}`\lambda f.e_2^*\, `\mathit{bind}`\, f :: m\, \tau_1}\ (\text{M-App})$$

However, without subtyping, the type of functions will be rather complex. For example, the following function:

```
until p f x = if p x then x else until p f (f x)
```

has the following type:

```
until :: Monad m => m ((a -> m Bool) -> m ((a -> m a) -> m (a -> m a)))
```

This type is a little bit complex and does not reflect the fact that supplying the first and the second argument of `until` causes no side-effect. Therefore, a desirable type would be something like:

```
until :: Monad m => (a -> m Bool) -> (a -> m a) -> a -> m a
```

In order to obtain such finer type, we will extend the former proposal [4] — the technique of subtyping and simplification of mixed type constraints — to subtyping between type constructor variables as type classes are extended to constructor classes [3]. Especially, we treat monads as extended polymorphic variants.

In [4], the author proposed a type inference system which separates subtyping and *matching* (record polymorphism), following to Bruce *et al*'s proposal. 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. While 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 is required. Matching is weaker than subtyping. That is, if σ has a *binary method*, then even when τ matches σ, τ is not a subtype of σ. Binary methods are methods which take another object of the same class as an argument [1].

Like most existing proposals of a type inference system for object-oriented features, we use a form of *constrained types*. A constrained type is a pair of a usual type expression and a set of type constraints. In the study of polymorphic record access [8, 11, *etc.*], constraints of the form $\alpha\#\{l :: \tau\}$ are used. It means that a type α must be a record with at least a label l of type τ. While, in the study of subtyping, constraints are 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. Polymorphic record access means that we can define functions which access the same fields of more than one kind of record types. It is basically type inference for matching — though existing proposals do not mention recursive records, it seems relatively easy to add such a feature. 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, type inference for subtyping suffers 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. In [2], Fuh and Mishra proposed rules to simplify subtyping constraints which are otherwise very complicated. Pottier [10] used a form of recursive subtyping constraints which also support record polymorphism. He defined a powerful entailment relation between recursive constraint sets and showed that much 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.

The point of the system in [4] is to separate subtyping and matching and to use both forms of type constraints. This separation not only makes it possible to avoid the problem of binary methods, but also makes the simplification of type constraints easy. The proposed system requires that the types of record labels and variant tags should be declared explicitly by the programmer, especially when they are recursive. The reason of this decision is as follows. First, the simplification of type constraints becomes much easier — since subtyping constraints are no longer used in order to unfold recursive types, the complexity of simplification is considerably reduced. Second, it is a straightforward extension of the current `data` or `datatype` declaration. And finally, we need type declaration of labels and tags anyway, when we extend the system to subtyping between type constructor variables as explained later.

In this system, a record type is declared as follows:

```
record Stream a = {head :: a, tail :: MyType}
```

By this declaration, two selector function `head` and `tail` become available. We can use such selectors to define functions as follows:

```
nthHead n xs = if n==0 then head xs else nthHead (n-1) (tail xs)
nthTail n xs = if n==0 then xs      else nthTail (n-1) (tail xs)
```

And the typing system simplifies their types to:

```
nthHead :: Int -> Stream a -> a
nthTail :: z # Stream a => Int -> z -> z
```

where `z # Stream a` means `z` is a record type which has two labels `head` and `tail` of appropriate types. Note that the type of `nthHead` is simplified so that no type constraint is left.

The rest of this paper is organized as follows. Section 2 gives a brief overview of the previous work of the author [4]. It explains our language — type constrains, typing rules, entailment relation. Simplification rules are given in Section 2.7. There, we also give some examples to show how type constraints are simplified. Section 3 extends the system to subtyping between type constructors and show some examples. Section 4 concludes.

2 Overview

In this section, we explain briefly the previously proposed system [4] before extending it to type constructor variables, though there are some improvement in presentation as well as new examples,

2.1 Declaration

We declare record labels as follows:

```
record Stream a = {head :: a, tail :: MyType}
```

By this declaration, two labels `head` and `tail` become available. For convenience, we give a name to each set of record constraints such as `Stream` above and use `Stream a` as a shorthand of a longer type expression `{head :: a, tail :: MyType}`.

Then `head`[1] has type `c # Stream a => c -> a`[2] and can be used as a selector function for records which have label `head` and `tail`. (As will be shown later, using our simplification rule, its type can be simplified to `Stream a -> a`.) Recursive labels are declared using the keyword `MyType`. Then, `tail` has type `c # Stream a => c -> c`. The type constraint `c # Stream a` means that `c` is a record type which has at least two labels `head` and `tail` with types `a` and `c` respectively. Such type constraints are added to the constraint set when we use selector functions such as `head`. We write type constraints in the left-hand-side of `=>`, following the Haskell syntax. As can be seen from the examples above, record constraints, in general, have type parameters (e.g. `a` in `c # Stream a`). Note that when a type has type constraints of the same label with different parameters, the corresponding parameters must be unified. For example, if `c` is constrained as `c # Stream a, c # Stream b, ...` then

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

[2] We use a Haskell-like syntax here, since we extend subtyping to type constructors later and the Haskell syntax is appropriate for this purpose — in the previous paper, this type is written as $c \rightarrow a \mid c\#Stream(a)$.

two parameters `a` and `b` 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. `c` above) as *independent* parameters and parameters after `#` (e.g. `a` and `b` above) as *dependent* parameters.

What is important is that we can define coercions in a calculus without subtyping into a type from any types that match that type, provided that `MyType` appears only positively in the definition. We say a type variable appears *positively* if it appears on the left-hand-side of an *even* number of arrows. And a type variable is said to appear *negatively* if it appears on the left-hand-side of an *odd* number of arrows.

Similarly, we can declare polymorphic variant tags in a way similar to ordinary `data` declarations as follows:

```
variant List a  = nil | cons a MyType
```

Then two "constructor" functions are defined — `nil` has type `c # Nil => c` and `cons` has type `c # Cons a => a -> c -> c`. They are used as "tags."

2.2 Expressions

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

$$e ::= x \;||\; \lambda x.e \;||\; e\,e \;||\; \texttt{let}\ x = e\ \texttt{in}\ e$$

The type language is usual except that it has type constraints. 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 $C \Rightarrow \tau$ 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.

2.3 Typing Rule

The typing rule of our language is given as follows:

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

$$\frac{A; x :: \tau \vdash e :: C \Rightarrow \tau'}{A \vdash \lambda x.e :: C \Rightarrow \tau \to \tau'}\text{(Lambda)} \qquad \frac{A \vdash e_1 :: C_1 \Rightarrow \tau_1 \to \tau \quad A \vdash e_2 :: C_2 \Rightarrow \tau_1}{A \vdash e_1\, e_2 :: C_1 \cup C_2 \Rightarrow \tau}\text{(App)}$$

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

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

In (Subtype), we use the entailment relation ($\Vdash$), which will be explained later.

Constructs for building and accessing pairs and records (and variants) are all viewed as primitive functions as in [10].

A set of type constraints would grow rapidly during type inference. In section 2.7, we will explain how such constraints are *simplified.*

2.4 Closure

After assembling type constraints, we calculate the *closure* of subtyping constraints. A constraint set C is *closed* if and only if the following conditions (+ conditions which will be explained shortly) hold.

$$\begin{array}{lrcl}
\text{(trans)} & \tau_1 \rhd \alpha \in C \wedge \alpha \rhd \tau_3 \in C & \Rightarrow & \tau_1 \rhd \tau_3 \in C \\
\text{(arrow)} & \sigma_1 \to \sigma_2 \rhd \tau_1 \to \tau_2 \in C & \Rightarrow & \tau_1 \rhd \sigma_1 \in C \wedge \sigma_2 \rhd \tau_2 \in C \\
\text{(struct)} & t\,\overline{\sigma}^p\,\overline{\sigma}^n \rhd t\,\overline{\tau}^p\,\overline{\tau}^n \in C \wedge \mathit{NoBinary}(t) & \Rightarrow & \sigma_i^p \rhd \tau_i^p \in C \wedge \sigma_i^n \lhd \tau_i^n \in C
\end{array}$$

($\overline{\tau}^p$ stands for a sequence of positive parameters $\tau_1^p, \ldots, \tau_n^p$ and $\overline{\tau}^n$ stands for a sequence of negative parameters $\tau_1^n, \ldots, \tau_n^n$. We write $\mathit{NoBinary}(t)$ to indicate that t have no binary methods (*i.e. MyType* does not occur 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(variant) type, we reduce it into subtyping relations between its parameters. Moreover, $(\alpha \to \beta) \rhd \gamma$ would be reduced into $\alpha' \rhd \alpha$, $\beta \rhd \beta'$ substituting γ with $\alpha' \to \beta'$. We do not take the "lazy" approach as in [10] since subtyping constraints are never recursive. If *MyType* has a negative occurrence in the definition of t, there is no subtype nor supertype of $t\,\overline{\sigma}$. This means that if we have $t\,\overline{\sigma} \rhd \alpha$ (or $\alpha \rhd t\,\overline{\sigma}$), α and $t\,\overline{\sigma}$ must be unified. Another way to say this is that if *MyType* has a negative occurrence in the definition of t, all the parameters of t are treated as if they appeared both positively and negatively in the definition of t. In the following, we will assume data types appearing in typing rules do not have binary methods.

When the supertype parameter of a subtyping relation is a record type ($\gamma \rhd r\,\overline{\tau}$)[3], it will be decomposed into record constraints and subtyping relations between parameters. For example,$\gamma \rhd \mathit{Stream}\,\alpha$ would be decomposed into $\gamma\#\mathit{Stream}\,\beta$ and $\beta \rhd \alpha$. 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.

In the opposite case, that is, when the subtype parameter of a subtyping constraint is a record type but the supertype is a variable ($r\,\overline{\sigma} \rhd \gamma$), the constraint is left as it is. That is, record constraints do *not* propagate *forward.* Moreover, if we have two constraints $\tau \rhd \alpha$ and $\alpha\#r\,\overline{\sigma}^p\,\overline{\sigma}^n$ in C, the record constraint propagate back as $\tau\#r\,\overline{\beta}^p\,\overline{\beta}^n$ and $\beta_i^p \rhd \sigma_i^p, \beta_i^n \lhd \sigma_i^n$. And of course, there are corresponding rules for variants. Therefore, we add the following to the

[3] 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.

definition of closed constraint sets.

$$\begin{array}{lll}
\text{(record1)} & \gamma \rhd r\,\overline{\sigma}^p\,\overline{\sigma}^n \in C & \Rightarrow\ \gamma\#r\,\overline{\alpha}^p\,\overline{\alpha}^n \in C \wedge \alpha_i^p \rhd \sigma_i^p \in C \wedge \alpha_i^n \lhd \sigma_i^n \in C \\
\text{(record2)} & \tau \rhd \alpha \in C \wedge \alpha\#r\,\overline{\sigma}^p\,\overline{\sigma}^n \in C & \Rightarrow\ \tau\#r\,\overline{\beta}^p\,\overline{\beta}^n \in C \wedge \beta_i^p \rhd \sigma_i^p \in C \wedge \beta_i^n \lhd \sigma_i^n \in C \\
\text{(variant1)} & v\,\overline{\sigma}^p\,\overline{\sigma}^n \rhd \gamma \in C & \Rightarrow\ \gamma\#v\,\overline{\beta}^p\,\overline{\beta}^n \in C \wedge \sigma_i^p \rhd \beta_i^p \in C \wedge \sigma_i^n \lhd \beta_i^n \in C \\
\text{(variant2)} & \alpha \rhd \tau \in C \wedge \alpha\#v\,\overline{\sigma}^p\,\overline{\sigma}^n \in C & \Rightarrow\ \tau\#v\,\overline{\beta}^p\,\overline{\beta}^n \in C \wedge \sigma_i^p \rhd \beta_i^p \in C \wedge \sigma_i^n \lhd \beta_i^n \in C
\end{array}$$

We check then that constraints generated by structural propagation and by transitive closures do not contain inconsistent constraints — constraints with incompatible toplevel type constructors such as $(\alpha, \beta) \rhd \gamma \to \delta$ and $(\alpha \to \beta)\#\mathit{Stream}\,\gamma$. We say a constraint set C is *consistent* if and only if it does not contain any inconsistent constraint. A constraint is *consistent* if and only if it belongs to one of the following forms. $\tau\#r\,\overline{\sigma}$ (if τ has methods required by $r\,\overline{\sigma}$), $\tau\#v\,\overline{\sigma}$, (if τ has methods required by $v\,\overline{\sigma}$), $\alpha \rhd \beta$, $t\,\overline{\sigma} \rhd t\,\overline{\tau}$, $\alpha \rhd r\,\overline{\sigma}$, $\tau \rhd r\,\overline{\sigma}$ (if $\tau\#r\,\overline{\sigma}$ is consistent), $r\,\overline{\sigma} \rhd \alpha$, $v\,\overline{\sigma} \rhd \alpha$, $v\,\overline{\sigma} \rhd \tau$ (if $\tau\#v\,\overline{\sigma}$ is consistent), $\alpha \rhd v\,\overline{\sigma}$

We must also do "occur check" here so that subtyping of the form $\alpha\#\tau[\alpha]$, $\alpha \rhd \tau[\alpha]$ or $\tau[\alpha] \rhd \alpha$ are not used, where $\tau[\alpha]$ is some type expression containing α. For example, a type constraint, $\alpha \rhd (\alpha \to \beta)$ 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 ($\gamma\#r[\alpha]$, $r[\alpha] \rhd \gamma$, $\gamma\#v[\alpha]$ and $\gamma \rhd v[\alpha]$) is found in the constraint set.) Otherwise, they have a possibility to make a cycle.

At this stage, remaining atomic constraints are either record (variant) constraints or subtyping constraints of the form $\alpha \rhd \beta$, $r\,\overline{\tau} \rhd \beta$ and $\alpha \rhd v\,\overline{\tau}$ 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*.

2.5 Entailment Relation

The entailment relation is defined as follows. $C \Vdash C'$ reads "C *entails* C'. The (RECORD) rule states that a type that matches a record type can be coerced into it, while the (VARIANT) rule says that a variant type can be coerced to any type that matches its corresponding variant constraint. Note that we simply write $C \Vdash c$ instead of $C \Vdash \{c\}$, when the right-hand-side of $\Vdash$ is a singleton.

$$\text{(AXIOM)}\ \frac{c \in C^\infty}{C \Vdash c} \qquad \text{(REFLEX)}\ \ C \Vdash \tau \rhd \tau \qquad \text{(SET)}\ \frac{\forall c \in C'.\ C \Vdash c}{C \Vdash C'}$$

$$\text{(ARROW)}\ \frac{C \Vdash \{\tau_1 \rhd \sigma_1, \sigma_2 \rhd \tau_2\}}{C \Vdash \sigma_1 \to \sigma_2 \rhd \tau_1 \to \tau_2} \qquad \text{(STRUCT)}\ \frac{C \Vdash \{\sigma_i ?_{t,i} \tau_i\}}{C \Vdash t\,\overline{\sigma} \rhd t\,\overline{\tau}}$$

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

Lemma:
The entailment relation ($\Vdash$) is transitive and reflexive.
Proof:
Reflexivity ($C \Vdash C$) is trivial. Transitivity ($C_1 \Vdash C_2 \land C_2 \Vdash C_3 \Rightarrow C_1 \Vdash C_3$) is proved by the induction on the structure of the derivation of $C_2 \Vdash C_3$. ■

2.6 Algorithmic Typing Rule

The typing rule introduced in Section 2.3 is not deterministic — it may assign multiple types to a single expression and not appropriate for type inference.

Here, we give the deterministic (algorithmic) typing rule and show that it is sound and complete with respect to the original typing rule.

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

$$\frac{A; x :: \tau \vdash^a e :: C \Rightarrow \tau'}{A \vdash^a \lambda x.e :: C \Rightarrow \tau \to \tau'}(\text{Lambda})^a \qquad \frac{A \vdash^a e_1 :: C_1 \Rightarrow \tau_1 \to \tau \quad A \vdash^a e_2 :: C_2 \Rightarrow \tau_2}{A \vdash^a e_1\, e_2 :: C_1 \cup C_2 \cup \{\tau_2 \rhd \tau_1\} \Rightarrow \tau}(\text{App})^{\text{a}}$$

$$\frac{\begin{array}{c}A \vdash^a e_1 :: C_1 \Rightarrow \tau_1 \quad A; x :: \forall\overline{\alpha}.C_1 \Rightarrow \tau_1 \vdash^a e_2 :: C_2 \Rightarrow \tau_2 \\ \overline{\alpha} = \text{FV}(C_1 \Rightarrow \tau_1) \setminus \text{FV}(A)\end{array}}{A \vdash^a \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 :: C_2 \Rightarrow \tau_2}(\text{Let})^a$$

That is, we use (Subtype) only in the right-hand-side of (App) rule.
Lemma: *(Soundness)*
If $A \vdash^a e :: C \Rightarrow \tau$, then $A \vdash e :: C \Rightarrow \tau$.
Proof: Each step in the deterministic version can be simulated by a step or two in the non-deterministic version. ■
Lemma: *(Minimal Typing Property)*
If $A \vdash e :: C \Rightarrow \tau$, then $A \vdash^a e :: C' \Rightarrow \tau'$ where $C \Vdash \tau' \rhd \tau$ and $C \Vdash C'$.
Proof: By induction on the structure of the derivation and a case analysis on the final rule used.
■

This lemma intuitively means that it sufficient to insert coercion only when functions are applied to arguments.

What we want to show next is that our type system is sound with respect to the operational semantics. (Well-typed programs do not go wrong.)

We must show that the type is preserved during the evaluation *(Subject reduction)* — if $e_1 :: \tau$ and $e_1 \longrightarrow e_2$, then $e_2 :: \tau$. Once we proved the Minimal Typing Lemma, the proof is standard using the "substitution lemma."

Of course, this property largely depends on "δ-typability."
Definition (δ-typability) Let k be a primitive function. If the type of k instantiates to $C \Rightarrow \tau' \to \tau$ and $v :: C' \Rightarrow \tau'$ where $(C \cup C')^\infty$ consistent, then $\delta(k, v)$ is defined and $\delta(k, v) :: \tau$.

We plan to check this in the future by translating the calculus into a more elementary one following Ohori's translation of record calculus.

2.7 Simplification Rules

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

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

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 the type "without fear of failure in the future." From the minimal typing lemma, this is also true for the "algorithmic version." 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. *(Cycle)*
- 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, (provided that their dependent parameters are the same.) *(UniqueBound)*

Actually, these two rules have been already used in [2].

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 must 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 the record type.

$$\text{(RecSubst)}\quad \frac{\begin{array}{c}A \vdash e :: C \Rightarrow \tau \quad \gamma \# r\,\overline{\sigma} \in C \quad \gamma \notin FV(A) \quad \not\exists \delta.\gamma \triangleright \delta \in C \\ \gamma \text{ appears only negatively in } \tau\end{array}}{A \vdash e :: (C \Rightarrow \tau)[r\,\overline{\sigma}/\gamma]}$$

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 given record type.

This rule is, of course, 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.

And the corresponding rule for variants is:

- 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 variant type.

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

We do not insist completeness here. That is, there may be cases where the four rules above are not applicable but we can find a substitution which satisfies the condition of (Subst) rule. However, it seems that in most cases, these for rules can make type constraints simple enough.

2.8 Examples

For example, `nthHead` and `nthTail` explained in the introduction:

```
nthHead n xs = if n==0 then head xs else nthHead (n-1) (tail xs)
nthTail n xs = if n==0 then xs      else nthTail (n-1) (tail xs)
```

have the following types before simplification.

$$\begin{array}{rcl} \mathit{nthHead} & :: & \left\{ \begin{array}{c} x \triangleright \mathit{Int}, \mathit{Int} \triangleright x, a \triangleright z, e \triangleright y, y \triangleright e, e \triangleright d, \\ e \# \mathit{Stream}\ a, d \# \mathit{Stream}\ a \end{array} \right\} \Rightarrow x \rightarrow y \rightarrow z \\ \mathit{nthTail} & :: & \left\{ \begin{array}{c} x \triangleright \mathit{Int}, \mathit{Int} \triangleright x, e \triangleright y, y \triangleright e, e \triangleright z, \\ e \# \mathit{Stream}\ a \end{array} \right\} \Rightarrow x \rightarrow y \rightarrow z \end{array}$$

But they are simplified to

$$\begin{array}{rcl} \mathit{nthHead} & :: & \mathit{Int} \rightarrow \mathit{Stream}\ a \rightarrow a \\ \mathit{nthTail} & :: & \{z \# \mathit{Stream}\ a\} \Rightarrow \mathit{Int} \rightarrow z \rightarrow z \end{array}$$

In the type of `nthHead`, no type constraint is left. While the type of `nthTail` still has a type constraint $z \# \mathit{Stream}\ a$. This is because z appears both positively and negatively in the type expression. If the type of `nthTail` were simplified to $\mathit{Int} \rightarrow \mathit{Stream}\ a \rightarrow \mathit{Stream}\ a$, it could not handle properly records with labels other than `head` and `tail`.

The following examples `map` and `concat`

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

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

have types before simplification:

$$\begin{array}{rcl} \mathit{map} & :: & \{c \triangleright x, \mathit{List}\ x \rightarrow \mathit{List}\ c, w \# \mathit{List}\ y\} \Rightarrow (x \rightarrow y) \rightarrow \mathit{List}\ c \rightarrow w \\ \mathit{concat} & :: & \{a \triangleright e, y \triangleright z, z \triangleright d, d \triangleright z, d \# \mathit{List}\ e\} \Rightarrow \mathit{List}\ a \rightarrow y \rightarrow z \end{array}$$

and types after simplification:

$$\begin{aligned} \mathit{map} &:: (x \to y) \to \mathit{List}\ x \to \mathit{List}\ y \\ \mathit{concat} &:: z\#\mathit{List}\ a \Rightarrow \mathit{List}\ a \to z \to z \end{aligned}$$

The type of `concat` has a type constraint $z\#\mathit{List}\ a$ for the same reason as `nthTail`.

3 Higher-Order Extension

In this section, we extend the system presented so far to simplify type expressions which typically arise when we write monadic programs, especially when we use the automatic translation of expressions to monadic form explained in the introduction.

Basically, monads can be seen as a variant type which has constructors corresponding to *return (unit)* and *then (bind)* operators. Its only peculiar point is that the type parameter changes in its recursive occurrence. Therefore, we invent new forms of declarations `variantClass` and `recordClass`.

```
variantClass m # Monad where
   return :: a -> m a
   (>>=)  :: m b -> (b -> m a) -> m a
```

Then, `return` has type `m # Monad => a -> m a`. The only difference with the ordinary `class` declaration is that, in `variantClass` declaration, the type constructor being defined must appear as the return type. If the constrained type has no varying parameters, we can use simple `variant` declaration.

In general, each monad has other constructors specific to it. For example, the state transformer monad [6] has two constructors below:

```
variantClass m s # Monad => m # StateMonad where
   readVar  :: MutVar s a -> m s a
   writeVar :: MutVar s a -> a -> m s ()
```

(Since the first parameter of `m` (`s`) is fixed in the definition, it might be possible to treat `s` as a simple parameter:

```
variantClass m' # Monad => m' # StateMonad s where
   readVar  :: MutVar s a -> m' a
   writeVar :: MutVar s a -> a -> m' ()
```

where `s` is treated as a dependent parameter of `m'` ($\approx$ `m a`).)

In order to define the corresponding data type for such variant (record) classes, we will need *existential types*.

```
data Monad a =  Return a | Then (Monad b) (b -> Monad a)
```

It is unusual since `Monad b` occurs in the definition of `Monad a` and that `b` must be existentially quantified. We assume that such data types are defined automatically with the corresponding **variantClass** declaration.

Accordingly, we must extend the subtyping relation to constructor variables such as `m` above. And, we must change the definition of the closure and the entailment relation to support this change. First, we extend the notion of *closed* constraint set.

$$\text{(cVar)} \quad \begin{array}{l} m\,\overline{\tau}^p\,\overline{\tau}^n \triangleright m'\,\overline{\sigma}^p\,\overline{\sigma}^n \in C \wedge m\#t\,\overline{\pi} \in C \wedge m'\#t\,\overline{\rho} \in C \\ \qquad \Rightarrow\ m \triangleright m' \in C \wedge \tau_i^p \triangleright \sigma_i^p \in C \wedge \tau_i^n \triangleleft \sigma_i^n \in C \end{array}$$

We must also add the following rule to the entailment relation:

$$\text{(CSTRUCT)} \quad \frac{C \Vdash \{m \triangleright m', \tau_i^p \triangleright \sigma_i^p, \tau_i^n \triangleleft \sigma_i^n\}}{C \Vdash \{m\,\overline{\tau}^p\,\overline{\tau}^n \triangleright m'\,\overline{\sigma}^p\,\overline{\sigma}^n\}}$$

In general, a type variable may have more than one type constraint that is declared by `variantClass` (`recordClass`) declaration. Then, a type variable can be constrained by more than one higher-order constraint. For example, there may be a case where a type variable a must be unified with `m1 x` where `m1#Foo` and also with `m2 y` where `m2#Bar`. In the current system, two type variables `x` and `y` must be unified, though they are essentially distinct. There would be several possibilities to fix this. However, we do not discuss it here, for such an extension is not necessary to treat monadic programs.

Then the simplification of higher-kind mixed type constraints becomes possible as well as first-order ones. Rules such as (Cycle), (UniqueBound), (RecSubst), (VarSubst) can be naturally extended to constructor variables.

For example, the following function:

```
incrVar r n = writeVar r (readVar r + n)
```

has the following type before simplification:

$$\begin{array}{ll} \mathit{incrVar} :: & \left\{ \begin{array}{c} m_1 \triangleright m_5, \mathit{StateMonad}\ s \triangleright m_2, m_2 \triangleright m_3, m_3 \triangleright m_4, m_4 \triangleright m_5, \\ m_1\#\mathit{Monad}, i \triangleright \mathit{Int}, a \triangleright \mathit{Int}, \mathit{Int} \triangleright a \end{array} \right\} \\ & \Rightarrow \mathit{MutVar}\ s\ a \to i \to m_5\ () \end{array}$$

By applying simplification rules, it simplifies to the following:

$$\mathit{incrVar} :: \mathit{MutVar}\ s\ \mathit{Int} \to \mathit{Int} \to \mathit{StateMonad}\ s\ ()$$

Here, we use the following rule to translate and type the expression.

$$\frac{A \vdash e_1 \leadsto e_1^* :: C_1 \Rightarrow m_1\,(\tau_1 \to m\,\tau) \quad A \vdash e_2 \leadsto e_2^* :: C_2 \Rightarrow m_2\,\tau_2}{A \vdash e_1\,e_2 \leadsto e_1^*\,`\mathit{bind}`\lambda f.e_2^*\,`\mathit{bind}`\ f :: C_1 \cup C_2 \cup \{\tau_2 \triangleright \tau_1, m_1 \triangleright m_3, m \triangleright m_3, m_2 \triangleright m_3\} \Rightarrow m_3\,\tau}\text{(M-App)}'$$

The type of `until`:

```
until f p x = if p x then x else until f p (f x)
```

before simplification is:

$$\mathit{until} \;::\; \left\{ \begin{array}{c} m\#\mathit{Monad}, \\ m_1 \triangleright n_2, n_1 \triangleright n_2, n_2 \triangleright m_5, m_5 \triangleright n_2, m_3 \triangleright m_5, \\ x_3 \triangleright \mathit{Bool}, x_1 \triangleright x_4, x_4 \triangleright x_0, x_4 \triangleright x_2, x_4 \triangleright x_5 \end{array} \right\}$$
$$\Rightarrow (x_0 \to m_1\, x_1) \to m_2\, ((x_2 \to m_3\, x_3) \to m_4\, (x_4 \to m_5\, x_5))$$

After simplification, it becomes as follows:

$$\mathit{until} \;::\; \{m_5\#\mathit{Monad}\} \Rightarrow (x_4 \to m_5\, x_4) \to \mathit{Monad}\, ((x_4 \to m_5\, \mathit{Bool}) \to \mathit{Monad}\, (x_4 \to m_5\, x_4))$$

If we adopt the convention that $a \to \mathit{Monad}\ b$ should be simply written as, for example, $a \rightarrowtail b$, the type of `until` is simply written as:

$$\mathit{until} \;::\; \{m_5\#\mathit{Monad}\} \Rightarrow (x_4 \to m_5\, x_4) \rightarrowtail (x_4 \to m_5\, \mathit{Bool}) \rightarrowtail (x_4 \to m_5\, x_4)$$

Of course, such expressions are mere variant data types as they are. Therefore, we have to give interpretation of constructors such as `(>>=)` and `return`. This would possibly be done by a mechanism similar to the usual `instance` declarations.

For a lazy language like Haskell, this extension allows programmers to write imperative programs in a more traditional and natural syntax. It even has a possibility to allow us to overload strict and lazy functions — the strict version can be obtained just by interpreting the monad as the strictness monad or the monad of continuations [13].

4 Conclusion

We studied type inference for a calculus with implicit subtyping and polymorphic records and variants and extended it to type constructor variables.

In practice, the calculus would be implemented by translation à la Ohori [8] into a calculus without implicit subtyping and polymorphic record and variant operations. Formalizing such a translation and proving its correctness is left as a future work as well as efficiency consideration of type inference.

Application of our higher-order extension to monadic programming is very similar to the polymorphic effect system [7, 12]. Currently, our system does not have sub-regioning. However, we expect that it can be incorporated into our system by using the notion of ***compositional references*** [5]. Moreover, in our system, programmers can define their own side-effects.

Unfortunately, it is likely that we cannot expect so much efficiency for such overloaded monadic programs. Probably, we will need a kind of a Just-In-Time compiler to get a reasonable efficiency.

Acknowledgment

I would like to thank Jacques Garrigue for the arrangement of the Workshop and other TTP Kyoto '97 participants for their valuable comments.

References

[1] Kim Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[2] 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.

[3] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *FPCA 93*. Springer Verlag, 1993.

[4] Koji Kagawa. Type inference for the mixture of matching and implicit subtyping. In *Proc. of the Second Fuji International Workshop on Functional and Logic Programming.* World Scientific Publishing, November 1996.

[5] Koji Kagawa. Compositional references for stateful functional programming. In *Proc. of the International Conference on Functional Programming 1997.* ACM Press, June 1997.

[6] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[7] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Annual ACM Symp. on Principles of Prog. Languages*, pages 47–57, 1988.

[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] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[13] Philip Wadler. Comprehending monads. In *ACM Symp. on Lisp and Functional Programming*, pages 61–78, 1990.

Other papers presented at the Workshop

For technical reasons, some papers cannot be included in these proceedings. We give here their references for an easy access.

References

[1] F. Damiani, M. Dezani-Ciancaglini, and P.Giannini. A filter model for mobile processes. *Mathematical Structures in Computer Science*, to appear. Available at `ftp://lambda.di.unito.it/pub/dezani/tesi.ps.gz`.

[2] A. Barber, P.A. Gardner, M. Hasegawa, and G. Plotkin. From action calculi to linear logic. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL)*, Aahrus, August 1997.

[3] Giorgio Ghelli. Complexity of kernel Fun subtype checking. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*, pages 134–145, Philadelphia, Pennsylvania, May 1996. ACM Press.

[4] Didier Rémy. From classes to objects via subtyping. In *Proceedings of the European Symposium on Programming (ESOP)*, April 1998. Available at `http://pauillac.inria.fr/~remy/publications.html`.

[5] Yasuhiko Minamide. A functional representation of data structures with a hole. In *Proceedings of the 25th ACM International Symposium on Principles of Programming Languages (POPL)*. ACM Press, January 1998. Available at `http://www.kurims.kyoto-u.ac.jp/~nan/hole.popl98.ps`.