数理解析研究所講究録
第851巻 1993年 21-33

A Complete Type Inference System for Subtyped Recursive Types

Tatsurou Sekiguchi and Akinori Yonezawa †
Department of Information Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan 113

Abstract. Since record polymorphism is one of essential factors for object-oriented languages, various approaches to incorporate record polymorphism into type systems have been made to lay the foundation for object-oriented languages. Recursive types, which are essentially types of lists or trees, are major programming tools. In object-oriented languages, a pseudo variable "self" has a recursive type, which requires that type systems be able to treat recursive types. The purpose of this paper is to provide a type system and its inference system which can handle both subtype polymorphism and recursive types without any kind of type declaration or unnatural restrictions. Our system integrates subtyping and parametric polymorphism into Damas and Milner's type system and preserves several properties such as existence of principle types and syntactic completeness of type checking algorithm. It can be also considered as [Cardelli,1984] added **let**. We give the type inference algorithm and prove its correctness. The basic idea is that we consider a type as a regular tree. Though our target language in this paper is a functional language, we show the way to extend it to imperative languages, too.

1. Introduction

Our aim is a largely practical one, that is to construct a sufficiently powerful type system and its feasible inference system for object-oriented languages. Since there are many discussions about what the definition of object-oriented languages is, we will not enter such a discussion. We only take account of existence of record types and polymorphic method selection. There are two sources of polymorphism – inheritance(subclassing)-based polymorphism and subtyping-based polymorphism. Although they seem to be the same thing, they are distinct as discussed in [Cook et al.,1990]. This is mainly caused by a difficulty of subtyping function space. Inheritance-based polymorphism is already implemented in many languages such as Smalltalk, C++, CLOS, Eiffel, etc. Subtyping-based polymorphism is proposed in [Cardelli,1984], which has simple and clear semantics and can avoid inheritance anomaly described in [Cook et al.,1990]. But the type checking algorithm seemed complicated so that it has not been implemented in practical languages. Recently, an attempt to incorporate subtyping-based polymorphism into C++ is made by [Baumgartner et al.,1992] although their mechanism is incomplete and fails to accept many subtypes since it judges a subtype relation only at one level of type constructors for efficiency.

The notion of recursive types appears naturally in many programming languages. A pair of recursive data structures and recursive functions over them is one of common programming techniques. A subtype relation over recursive types was defined by [Amadio et al.,1991]. Recursive types arise necessarily in object-oriented languages since a pseudo variable such as **this** or **self** which belongs to all records implicitly has a recursive type inevitably.

We have obtained a result that a complete inference system with product(record type) and sum(variant) in the presence of subtyping-based polymorphism, recursive types and parametric polymorphism is possible without any kind of type declaration (such as datatype in ML) or unnatural restrictions. Our system, so to speak, finds bounded quantification[Cardelli et al.,1985] by inference. We can construct this system by means of extending the notion of [Amadio et al.,1991] that considers a type as a regular tree. Our system is an extension of Damas and Milner's type system [Damas,1985] and preserves several properties such as existence of principle types. Soundness and completeness of our inference system are proved. From the point of our result, subtyping-based polymorphism is considered to be more natural than inheritance-based polymorphism. In our type system, the notion of safe operation coincides with polymorphism entirely so that it provides simple

†E-mail: {cocoa,yonezawa}@is.s.u-tokyo.ac.jp

semantics and full reusability of function. F-bounded polymorphism is proposed in [Canning et al.,1989]. Although they thought that F-bounded polymorphism was more than subtype relation, if types are defined in our way, they turn out to be equivalent to each other (see the end of § 4.2).

There are already several type systems that handle subtyping or recursive types. The type systems of [Cardelli et al.,1985] and [Canning et al.,1989] have both of them, but it follows that they are strongly typed languages so that a type of a variable is declared by the programmer. The system of [Wand,1987] has record subtyping without recursive types. The system of [Stansifer,1988] does not treat recursive types. But we can see a notion of lub and glb and it provides an algorithm that finds a principle type. Although [Kaes,1992] can handle recursive types, his system is too general for our purpose and gives no consideration to practical languages.

The rest of this paper is organized as follows. Section 2 defines regular types. The target language of our system is a simple λ-like language described in Section 3. Section 4 is the main part, which describes our type inference system and proves its soundness and completeness. An extension to imperative languages is also discussed in this section. Section 5 gives an example. The relationship between subtyping and subclassing is discussed in Section 6. Section 7 summarizes our study and indicates directions for future investigation.

2. Types

2.1. Regular Types. $\mathbf{N}$ denotes a set of all natural numbers including 0. Let T_P be a finite set of all primitive types such as $\mathbf{Bool}, \mathbf{Int}, \mathbf{Str}, \ldots$, T_V be a set of all type variables(ranged over $\alpha, \beta, \gamma, \ldots$), T_C be a set of type constructors, which consist of arrows $\rightarrow_2$, products $\{\}_n$, and sums $[]_n$ where subscripts are their arities for $n \in \mathbf{N}$, and finally T_L be a set of labels. We often use a type constructor $\rightarrow$ as a right associative infix operator. T_P, T_V, T_C and T_L are disjoint. We shall denote by $R(T)$ the set of regular trees over T where $T = (T_P, T_V, T_C, T_L)$. We use a rational expression to represent a regular tree. $\mathrm{RE}(T)$ is a set of all rational expressions over T, which is the least set defined as follows:

$$
\begin{array}{l}
T_P \subseteq \mathrm{RE}(T) \\
T_V \subseteq \mathrm{RE}(T) \\
\alpha, \beta \in \mathrm{RE}(T) \implies \alpha \rightarrow \beta \in \mathrm{RE}(T) \\
\alpha_1, \cdots, \alpha_n \in \mathrm{RE}(T) \implies \{l_1 : \alpha_1, \cdots, l_n : \alpha_n\} \in \mathrm{RE}(T) \text{ where } l_i \in T_L \text{ for } n \in \mathbf{N} \\
\alpha_1, \cdots, \alpha_n \in \mathrm{RE}(T) \implies [l_1 : \alpha_1, \cdots, l_n : \alpha_n] \in \mathrm{RE}(T) \text{ where } l_i \in T_L \text{ for } n \in \mathbf{N} \\
\tau \in T_V, \alpha \in \mathrm{RE}(T) \implies \mu\tau.\alpha \in \mathrm{RE}(T)
\end{array}
$$

A type expression is defined as the following rules.

$$\tau = \sigma|^{\forall}\alpha_i.\tau \qquad \sigma \in \mathrm{RE}(T) \text{ and } \alpha_i \in T_V$$

This is similar to ML's definition of types except that σ is a general rational expression. $\mu\tau.\alpha$ denotes a unique regular tree in $R(T)$ such that $\mu\tau.\alpha = \alpha[\mu\tau.\alpha/\tau]$ where [] is a substitution. The existence and uniqueness is explained in [Courcelle,1983]. A type is called closed if it has no free type variables and called ground if it has no type variables. A ground type is closed. We have two different ways to bind type variables – universal quantification and μ-recursion. Free type variables and type variables of a type expression are defined by the following functions FTV and TV, respectively.

$$
\begin{array}{ll}
\mathrm{TV}(\alpha) = \emptyset & \mathrm{FTV}(\alpha) = \emptyset \quad \text{if } \alpha \in T_P \\
\mathrm{TV}(\alpha) = \{\alpha\} & \mathrm{FTV}(\alpha) = \{\alpha\} \quad \text{if } \alpha \in T_V \\
\mathrm{TV}(\alpha \rightarrow \beta) = \mathrm{TV}(\alpha) \cup \mathrm{TV}(\beta) & \mathrm{FTV}(\alpha \rightarrow \beta) = \mathrm{FTV}(\alpha) \cup \mathrm{FTV}(\beta) \\
\mathrm{TV}((\alpha_1, \cdots, \alpha_n)) = \mathrm{TV}(\alpha_1) \cup \cdots \cup \mathrm{TV}(\alpha_n) & \mathrm{FTV}(\{\alpha_1, \cdots, \alpha_n\}) = \mathrm{FTV}(\alpha_1) \cup \cdots \cup \mathrm{FTV}(\alpha_n) \\
\mathrm{TV}([\alpha_1, \cdots, \alpha_n]) = \mathrm{TV}(\alpha_1) \cup \cdots \cup \mathrm{TV}(\alpha_n) & \mathrm{FTV}([\alpha_1, \cdots, \alpha_n]) = \mathrm{FTV}(\alpha_1) \cup \cdots \cup \mathrm{FTV}(\alpha_n) \\
\mathrm{TV}(\mu\tau.\alpha) = \mathrm{TV}(\alpha) & \mathrm{FTV}(\mu\tau.\alpha) = \mathrm{FTV}(\alpha) - \{\tau\} \\
\mathrm{TV}(^{\forall}\alpha.\tau) = \{\alpha\} \cup \mathrm{TV}(\tau) & \mathrm{FTV}(^{\forall}\alpha.\tau) = \mathrm{FTV}(\tau) - \{\alpha\}
\end{array}
$$

Example 1. $\mathrm{TV}(^{\forall}\alpha.\mu\tau.(\alpha \rightarrow \mathbf{Int}) \rightarrow (\beta \rightarrow \tau)) = \{\alpha, \beta\}$. $\mathrm{FTV}(^{\forall}\alpha.\mu\tau.(\alpha \rightarrow \mathbf{Int}) \rightarrow (\beta \rightarrow \tau)) = \{\beta\}$.

The rule to determine which μ-operator binds the type variable is as the same as usual λ-calculus. We call the variable μ-variable.

2.2. Product of Types. We will use various operations over types. The most fundamental one is a product of types. Note that product of types is not a product(record) type. Suppose α and β be types. Let $S_\alpha = [x_1 = u_1, \cdots, x_n = u_n]$ and $S_\beta = [y_1 = v_1, \cdots, y_m = v_m]$ be regular systems which denote regular types represented by α and β respectively where $x_i, y_j \in T_V$ and $u_i, v_j \in \mathrm{RE}(T)$ for all i and j. A straightforward translation algorithm of a regular system to a rational expression is to make all x_i to be μ-variables and substitute all subexpressions except u_1. For instance, S_α can be translated to $[\mu x_n.u_n/x_n] \cdots [\mu x_2.u_2/x_2]\mu x_1.u_1$

automatically. In this case, a capture of $x_1, \cdots, x_{n-1}$ is used on purpose. Therefore, we call $x_1, \cdots, x_n$ μ-variables of a regular system as a natural extension of terminology. We consider the first component of a regular system, say x_1 in S_α, as the regular type represented by the system. The product of α and β is a new regular type whose components are ordered pairs $\langle x, y \rangle$ where x, y are primitive types or type variables and generated by the following algorithm. An infix operator $\times$ denotes product operation.

1. Start with $x_1 \times y_1$.
2. $\alpha \times \beta = \langle \alpha, \beta \rangle$ if α and β are primitive types or free type variables.
3. If α or β is a μ-variable in $\alpha \times \beta$ and the pair is not compared yet, then we make two new μ-variables, say x' and y', and add the new equation $\langle x', y' \rangle = \alpha \times \beta$ to the new regular system. Then expand the μ-variable and continue the computation.
4. If one of the arguments is a μ-variable and the pair is already compared, then the result is the pair $\langle x', y' \rangle$ which is generated in the last time.
5. $(\alpha_1 \to \alpha_2) \times (\beta_1 \to \beta_2) = (\alpha_1 \times \beta_1) \to (\alpha_2 \times \beta_2)$.
6. $\{l_i : \alpha_i, m_j : \alpha'_j\} \times \{l_i : \beta_i, n_k : \beta'_k\} = \{l_i : \alpha_i \times \beta_i, m_j : \alpha'_j \times 0, n_k : 0 \times \beta'_k\}$. $(^\forall j, ^\forall k . m_j \neq n_k)$
7. $[l_i : \alpha_i, m_j : \alpha'_j] \times [l_i : \beta_i, n_k : \beta'_k] = [l_i : \alpha_i \times \beta_i, m_j : \alpha'_j \times 0, n_k : 0 \times \beta'_k]$. $(^\forall j, ^\forall k . m_j \neq n_k)$

where 0 is a special constant which compensates the absence of the other element. multiplication by 0 is defined as follows:

1. $0 \times \alpha = \langle 0, \alpha \rangle, \alpha \times 0 = \langle \alpha, 0 \rangle$ if α is a primitive type or a free type variable.
2. $(\alpha_1 \to \alpha_2) \times 0 = (\alpha_1 \times 0) \to (\alpha_2 \times 0), 0 \times (\alpha_1 \to \alpha_2) = (0 \times \alpha_1) \to (0 \times \alpha_2)$.
3. $\{l_i : \alpha_i\} \times 0 = \{l_i : \alpha_i \times 0\}, 0 \times \{l_i : \alpha_i\} = \{l_i : 0 \times \alpha_i\}$.
4. $[l_i : \alpha_i] \times 0 = [l_i : \alpha_i \times 0], 0 \times [l_i : \alpha_i] = [l_i : 0 \times \alpha_i]$.

Product of two types is almost as the same as unification operation over regular systems except that variables and constants are not unified. A comparison between different type constructors fails or the result is considered to be $\bot$.

Example 2. Suppose $\alpha = \mu\tau.\mathbf{Int} \to \tau$ and $\beta = \mu\sigma.\mathbf{Nat} \to \mathbf{Int} \to \sigma$. A regular system $S_\alpha = [x_1 = \mathbf{Int} \to x_1]$ denotes α and $S_\beta = [y_1 = \mathbf{Nat} \to y_2, y_2 = \mathbf{Int} \to y_1]$ denotes β. Then,

$$\begin{aligned} S_\alpha \times S_\beta &= [\langle x_1, y_1 \rangle = x_1 \times y_1] \\ &= [\langle x_1, y_1 \rangle = (\mathbf{Int} \to x_1) \times (\mathbf{Nat} \to y_2)] \\ &= [\langle x_1, y_1 \rangle = \langle \mathbf{Int}, \mathbf{Nat} \rangle \to (x_1 \times y_2)] \\ &= [\langle x_1, y_1 \rangle = \langle \mathbf{Int}, \mathbf{Nat} \rangle \to \langle x_1, y_2 \rangle, \langle x_1, y_2 \rangle = (\mathbf{Int} \to x_1) \times (\mathbf{Int} \to y_1)] \\ &= [\langle x_1, y_1 \rangle = \langle \mathbf{Int}, \mathbf{Nat} \rangle \to \langle x_1, y_2 \rangle, \langle x_1, y_2 \rangle = \langle \mathbf{Int}, \mathbf{Int} \rangle \to \langle x_1, y_1 \rangle] \end{aligned}$$

As well known in [Courcelle,1983], a regular system, a regular tree and a rational expression can be considered as equivalent each other. For instance, the above regular system is represented by a rational expression $\mu\tau.\langle \mathbf{Int}, \mathbf{Nat} \rangle \to \langle \mathbf{Int}, \mathbf{Int} \rangle \to \tau$. Therefore we shall identify them from now. The first projection π_1 and the second projection π_2 are defined naturally such that $\pi_1(\tau) \times \pi_2(\tau) = \tau$ for any type τ.

2.3. Subtype Relation over Infinite Types. In order to define a subtype relation over infinite types, we introduce an auxiliary notion, subproduct. A subproduct of types is very similar to a product of types except that the following rules are replaced with. We denote subproduct by $*$.

5′. $(\alpha_1 \to \alpha_2) * (\beta_1 \to \beta_2) = (\beta_1 * \alpha_1) \to (\alpha_2 * \beta_2)$.
6′. $\{l_i : \alpha_i, m_j : \alpha'_j\} * \{l_i : \beta_i\} = \{l_i : \alpha_i * \beta_i\}$.
7′. $[l_i : \alpha_i] * [l_i : \beta_i, n_k : \beta'_k] = [l_i : \alpha_i * \beta_i]$.

In the rule 1, α_1 and β_1 change their places unlike product since function is contravariant for its domain. A comparison between inadequate products or sums fails (for instance, $\{l_1 : \mathbf{Int}\} * \{l_1 : \mathbf{Int}, l_2 : \mathbf{Str}\}$). It is denoted by $\bot$ as the case of product of types. We define $\bot$ to be idempotent, i.e., $\bot * \alpha = \alpha * \bot = \bot$.

Example 3. Suppose $\alpha = \mathbf{Int} \to \{\text{foo} : \mathbf{Int}, \text{bar} : \mathbf{Bool}\}$ and $\beta = \mathbf{Nat} \to \{\text{foo} : \mathbf{Int}\}$,

$$\begin{aligned} \alpha * \beta &= (\mathbf{Int} \to \{\text{foo} : \mathbf{Int}, \text{bar} : \mathbf{Bool}\}) * (\mathbf{Nat} \to \{\text{foo} : \mathbf{Int}\}) \\ &= \langle \mathbf{Nat}, \mathbf{Int} \rangle \to (\{\text{foo} : \mathbf{Int}, \text{bar} : \mathbf{Bool}\} * \{\text{foo} : \mathbf{Int}\}) \\ &= \langle \mathbf{Nat}, \mathbf{Int} \rangle \to \{\text{foo} : \langle \mathbf{Int}, \mathbf{Int} \rangle\} \end{aligned}$$

$*$ is neither commutative nor associative.

A subtype relation over a domain of infinite types was defined by [Amadio et al.,1991] in case of function space, and they also gave an algorithm to determine a subtype relation for any two ground types and demonstrated its soundness and completeness. We can explain it in terms of a ranked subproduct and extend it to involve product(record type) and sum(variant). $*_n$ denotes the subproduct of n-th rank. Assume that a

subtype relation over primitive types is given. We shall denote a subtype relation by $\leq$. The ranked subproduct $*_n$ is a Boolean binary function similar to the subproduct operation except that:

1. $\alpha *_0 \beta$ is always true.
2. If both α and β are primitive types, then $\langle \alpha, \beta \rangle$ is true iff $\alpha \leq \beta$. Otherwise false.
3. $(\alpha_1 \rightarrow \alpha_2) *_n (\beta_1 \rightarrow \beta_2) = (\beta_1 *_{n-1} \alpha_1) \rightarrow (\alpha_2 *_{n-1} \beta 2)$.
4. $\{l_i : \alpha_i, m_j : \alpha'_j\} *_n \{l_i : \beta_i\} = \{l_i : \alpha_i *_{n-1} \beta_i\}$.
5. $[l_i : \alpha_i] *_n [l_i : \beta_i, n_k : \beta'_k] = [l_i : \alpha_i *_{n-1} \beta_i]$.

This function *cuts* a type at a depth of n and the rests are considered to satisfy the subtype condition. [Amadio et al.,1991] defines a subtype relation as follows:

$$^\forall k \in \mathbf{N}.\alpha *_k \beta \text{ is true} \implies \alpha \leq \beta$$

This relation induces partial order on the set of all ground types. If both α and β are regular types, then there is a finite n such that $\alpha *_n \beta \Longleftrightarrow {}^\forall k \in \mathbf{N}.\alpha *_k \beta$. We see this fact by giving an algorithm that must terminate to judge the subtype relation. To prove $\alpha \leq \beta$, compute $\alpha * \beta$ first. The number of all components is finite as noted above. If all components $\langle \alpha_i, \beta_j \rangle$ are true, then we conclude $\alpha \leq \beta$. This algorithm completely agrees with the above definition, obviously. The complexity is less than or equal to that of unification of regular trees. Therefore, in many cases, subtype relation can be judged in quasi-linear order for the lengths of type expressions.

Example 4. Suppose $\mathbf{Nat} \leq \mathbf{Int}$ and $\alpha = \mathbf{Int} \rightarrow \{\text{foo} : \mathbf{Int}, \text{bar} : \mathbf{Bool}\}$ and $\beta = \mathbf{Nat} \rightarrow \{\text{foo} : \mathbf{Int}\}$ are given. $\alpha * \beta = \langle \mathbf{Nat}, \mathbf{Int} \rangle \rightarrow \{\text{foo} : \langle \mathbf{Int}, \mathbf{Int} \rangle\}$. For $\langle \mathbf{Nat}, \mathbf{Int} \rangle$ and $\langle \mathbf{Int}, \mathbf{Int} \rangle$, $\mathbf{Nat} \leq \mathbf{Int}$ and $\mathbf{Int} \leq \mathbf{Int}$ hold. Thus, $\alpha \leq \beta$.

2.4. LUB Type and GLB Type. The notion of LUB(the Least Upper Bound) types and GLB(the Greatest Lower Bound) types can be already seen in [Cardelli,1984] and [Stansifer,1988]. Lub and glb of types are used to solve a set of subtype constraints. An upper bound of an unknown type is given by the greatest lower bound of its upper bound types. [Cardelli,1984] treated only ground types and an algorithm to judge a subtype relation between recursive types was obscure. Later, it became clear in [Amadio et al.,1991]. [Stansifer,1988] has no recursive types. Our definition is an extension of Cardelli's to regular types with quantified variables. Cardelli referred to them as join and meet types, respectively. According to his notation, we denote the lub type of α and β by $\alpha \uparrow \beta$ and the glb type by $\alpha \downarrow \beta$. To give their definition, we need some preliminary remarks.

A function is contravariant for its domain since $\alpha \rightarrow \beta \leq \alpha' \rightarrow \beta'$ implies $\alpha' \leq \alpha$ and $\beta \leq \beta'$. This contravariance has caused a lot of nuisances on constructing type systems and type inference systems with subtyping. A distinction between subtyping and subclassing is due to this fact. We introduce normalization of rational expressions to eliminate some problem related to contravariance. We say α is in a negative position if α is judged in a contravariant- way. Conversely, we say α is in a positive position if α is judged in a covariant(ordinary) way. In a rational expression that represents a regular type, the same primitive type or type variable may be referred to more than once. We say α is in an overlapping position if α is in both a positive position and a negative position.

Example 5. In a function type $\mathbf{Int} \rightarrow \mathbf{Nat}$, **Int** is in a negative position and **Nat** is in a positive position. As for $\mu\tau.\tau \rightarrow \mathbf{Int}$, **Int** is in an overlapping position.

An overlapping position causes some difficulty to compute lub and glb. Hence, we first eliminate those positions and transform types into an equivalent representation. We say a type is normal if its rational expression has no overlapping positions. It is done by the following algorithm. We denote a covariant type constructor by $\xrightarrow{+}$ and a contravariant constructor by $\xrightarrow{-}$. Let $\{q_i\}$ be a set of all positions. For instance, $\alpha \rightarrow \beta$ is denoted by $\alpha \xleftarrow{-} q_0 \xrightarrow{+} \beta$. First, we make new positions $(q_i, +)$ and $(q_i, -)$ for all q_i. If there is a covariant constructor such that $q_i \xrightarrow{+} q_j$, then connect positions to be $(q_i, +) \xrightarrow{+} (q_j, +)$ and $(q_i, -) \xrightarrow{+} (q_j, -)$. If there is a contravariant constructor such that $q_i \xrightarrow{-} q_j$, then connect positions to be $(q_i, +) \xrightarrow{-} (q_j, -)$ and $(q_i, -) \xrightarrow{-} (q_j, +)$. The resulting rational expression has no overlapping positions obviously and is as the same as the original one.

Example 6. $\mu\tau.\tau \rightarrow \mathbf{Int}$ is transformed into $\mu\tau.(\tau \rightarrow \mathbf{Int}) \rightarrow \mathbf{Int}$. The first **Int** is in a negative position and the second **Int** is in a positive position.

A lub type of two regular types is computed as the following algorithm. We assume that lubs and glbs of primitive types are defined. After eliminating all overlapping positions,

1. If either is a free type variable in $\alpha \uparrow \beta$, the result is a pair $\langle \alpha, \beta \rangle^\uparrow$.
2. $(\alpha_1 \rightarrow \alpha_2) \uparrow (\beta_1 \rightarrow \beta_2) = (\alpha_1 \downarrow \beta_1) \rightarrow (\alpha_2 \uparrow \beta_2)$.
3. $\{l_i : \alpha_i, m_j : \alpha'_j\} \uparrow \{l_i : \beta_i, n_k : \beta'_k\} = \{l_i : \alpha_i \uparrow \beta_i\}$. $\qquad ({}^\forall j, {}^\forall k.m_j \neq n_k)$

4. $[l_i : \alpha_i, m_j : \alpha'_j] \uparrow [l_i : \beta_i, n_k : \beta'_k] = [l_i : \alpha_i \uparrow \beta_i, m_j : \alpha'_j, n_k : \beta'_k]$. $(\forall j, \forall k. m_j \neq n_k)$

Rules about μ-variables are similar to ones of product of types. Comparison between different type constructors fails. We denote it by $\bot$ as the same as the case of product.

Example 7. Suppose that $\mathbf{Nat} < \mathbf{Int}$, $\alpha = \mu\tau.\mathbf{Int} \to \tau$ and $\beta = \mu\sigma.\mathbf{Nat} \to \mathbf{Int} \to \sigma$. Then,

$$\begin{aligned}
\alpha \uparrow \beta &= (\mu\tau.\mathbf{Int} \to \tau) \uparrow (\mu\sigma.\mathbf{Nat} \to \mathbf{Int} \to \sigma) \\
&= \mu\langle\tau,\sigma\rangle.(\mathbf{Int} \downarrow \mathbf{Nat}) \to (\tau \uparrow (\mathbf{Int} \to \sigma)) \\
&= \mu\langle\tau,\sigma\rangle.\mathbf{Nat} \to ((\mathbf{Int} \to \tau) \uparrow (\mathbf{Int} \to \sigma)) \\
&= \mu\langle\tau,\sigma\rangle.\mathbf{Nat} \to ((\mathbf{Int} \downarrow \mathbf{Int}) \to (\tau \uparrow \sigma)) \\
&= \mu\langle\tau,\sigma\rangle.\mathbf{Nat} \to (\mathbf{Int} \to \langle\tau,\sigma\rangle)
\end{aligned}$$

The case of glb is simply defined as this contravariance.

1'. If either is a free type variable in $\alpha \downarrow \beta$, the result is a pair $\langle\alpha,\beta\rangle^{\downarrow}$.
2'. $(\alpha_1 \to \alpha_2) \downarrow (\beta_1 \to \beta_2) = (\alpha_1 \uparrow \beta_1) \to (\alpha_2 \downarrow \beta_2)$.
3'. $\{l_i : \alpha_i, m_j : \alpha'_j\} \downarrow \{l_i : \beta_i, n_k : \beta'_k\} = \{l_i : \alpha_i \downarrow \beta_i, m_j : \alpha'_j, n_k : \beta'_k\}$. $(\forall j, \forall k. m_j \neq n_k)$
4'. $[l_i : \alpha_i, m_j : \alpha'_j] \downarrow [l_i : \beta_i, n_k : \beta'_k] = [l_i : \alpha_i \downarrow \beta_i]$. $(\forall j, \forall k. m_j \neq n_k)$

The problem caused by overlapping positions is that lub and glb operation may be reversed when a μ-variable is unfolded. We give an example.

Example 8. Suppose that $\mathbf{Nat} < \mathbf{Int}$, $\alpha = \mu\tau.\tau \to \mathbf{Nat}$ and $\beta = \mu\sigma.\sigma \to \mathbf{Int}$. **Nat** and **Int** are in overlapping positions. Then,

$$\begin{aligned}
\alpha \uparrow \beta &= (\mu\tau.\tau \to \mathbf{Nat}) \uparrow (\mu\sigma.\sigma \to \mathbf{Int}) \\
&= \mu\langle\tau,\sigma\rangle.(\tau \downarrow \sigma) \to (\mathbf{Nat} \uparrow \mathbf{Int}) \\
&= \mu\langle\tau,\sigma\rangle.\langle\tau,\sigma\rangle \to \mathbf{Int}
\end{aligned}$$

Although $\alpha' = \mu\tau.(\tau \to \mathbf{Nat}) \to \mathbf{Nat}$ and $\beta' = \mu\sigma.(\sigma \to \mathbf{Int}) \to \mathbf{Int}$ are equivalent representation of α and β, respectively, their lub is derived as follows:

$$\begin{aligned}
\alpha' \uparrow \beta' &= (\mu\tau.(\tau \to \mathbf{Nat}) \to \mathbf{Nat}) \uparrow (\mu\sigma.(\sigma \to \mathbf{Int}) \to \mathbf{Int}) \\
&= \mu\langle\tau,\sigma\rangle.((\tau \to \mathbf{Nat}) \downarrow (\sigma \to \mathbf{Int})) \to (\mathbf{Nat} \uparrow \mathbf{Int}) \\
&= \mu\langle\tau,\sigma\rangle.((\tau \uparrow \sigma) \to (\mathbf{Nat} \downarrow \mathbf{Int})) \to \mathbf{Int} \\
&= \mu\langle\tau,\sigma\rangle.(\langle\tau,\sigma\rangle \to \mathbf{Nat}) \to \mathbf{Int}
\end{aligned}$$

If there is no overlapping positions, no such phenomenon occurs obviously.

More generally, lub type and glb type can be defined over infinite ground types. For that, we need finite approximation of lub and glb and limit operation. We introduce a new type 0 into primitive types. A ranked lub $\uparrow_k$ ($k \in \mathbf{N}$) is defined as follows:

1. $\alpha \uparrow_0 \beta = \alpha \downarrow_0 \beta = 0$ for any types α, β.
2. $(\alpha_1 \to \alpha_2) \uparrow_k (\beta_1 \to \beta_2) = (\alpha_1 \downarrow_{k-1} \beta_1) \to (\alpha_2 \uparrow_{k-1} \beta_2)$.
3. $\{l_i : \alpha_i, m_j : \alpha'_j\} \uparrow_k \{l_i : \beta_i, n_k : \beta'_k\} = \{l_i : \alpha_i \uparrow_{k-1} \beta_i\}$. $(\forall j, \forall k. m_j \neq n_k)$
4. $[l_i : \alpha_i, m_j : \alpha'_j] \uparrow_k [l_i : \beta_i, n_k : \beta'_k] = [l_i : \alpha_i \uparrow_{k-1} \beta_i, m_j : \alpha'_j, n_k : \beta'_k]$. $(\forall j, \forall k. m_j \neq n_k)$

$\downarrow_k$ is defined similarly.

Definition 1 (lub on infinite ground types). Let α, β be infinite ground types.

$$\alpha \uparrow \beta = \lim_{k \to \infty} \alpha \uparrow_k \beta$$

A sequence of ranked lubs converges or a limit exists if $\alpha \uparrow_k \beta$ exists for all $k \in \mathbf{N}$. The domain of infinite ground types with subtyping order has several properties. It looks like a carved lattice. There are some pairs of elements which have no lub or glb elements. But if either exists, another must exist.

Proposition 1. *$\alpha \uparrow \beta \neq \bot$ if and only if $\alpha \downarrow \beta \neq \bot$.*

Proof. We have assumed that primitive types have this property. A necessary and sufficient condition of existence of lubs and glbs is to match kinds of type constructors for every position in two types. Hence, if either exists, it implies all type constructors match in view of a kind(that is, function, product or sum). ∎

The next theorem tells a uniqueness of lub and glb. This proposition is used to bundle several subtype constraints into one.

Proposition 2 (Uniqueness of lub and glb). *(i). $\uparrow$ is commutative. (ii). $\uparrow$ is associative.*

Proof. (i). Obvious. (ii). To exclude trivial cases, we assume that the lub and glb of every pair of types under consideration is not $\bot$.

Fact 1. *Let α, β and γ be infinite ground types. $(\alpha \uparrow_k \beta) \uparrow_k \gamma = \alpha \uparrow_k (\beta \uparrow_k \gamma)$ for all $k \in \mathbf{N}$.*

Proof. We show associativity of each rule. The proof is by induction.

Base case. We assume that $\uparrow_k$ and $\downarrow_k$ are associative for primitive types.

Case of arrows.

$$\begin{aligned}
&((\alpha_1 \to \alpha_2) \uparrow_k (\beta_1 \to \beta_2)) \uparrow_k (\gamma_1 \to \gamma_2) \\
=&((\alpha_1 \downarrow_{k-1} \beta_1) \to (\alpha_2 \uparrow_{k-1} \beta_2)) \uparrow_k (\gamma_1 \to \gamma_2) \\
=&((\alpha_1 \downarrow_{k-1} \beta_1) \downarrow_{k-1} \gamma_1) \to ((\alpha_2 \uparrow_{k-1} \beta_2) \uparrow_{k-1} \gamma_2) \\
=&(\alpha_1 \downarrow_{k-1} (\beta_1 \downarrow_{k-1} \gamma_1)) \to (\alpha_2 \uparrow_{k-1} (\beta_2 \uparrow_{k-1} \gamma_2)) \quad \text{(by IH)} \\
=&(\alpha_1 \to \alpha_2) \uparrow_k ((\beta_1 \downarrow_{k-1} \gamma_1) \to (\beta_2 \uparrow_{k-1} \gamma_2)) \\
=&(\alpha_1 \to \alpha_2) \uparrow_k ((\beta_1 \to \beta_2) \uparrow_k (\gamma_1 \to \gamma_2))
\end{aligned}$$

Case of product. Intersection is associative.

$$\begin{aligned}
&(\{l_i : \alpha_i, m_j : \alpha'_j\} \uparrow_k \{l_i : \beta_i, n_k : \beta'_k\}) \uparrow_k \{l_i : \gamma_i, o_p : \gamma'_p\} \qquad ({}^\forall j, {}^\forall k, {}^\forall p. m_j \neq n_k \neq o_p) \\
=&\{l_i : \alpha_i \uparrow_{k-1} \beta_i\} \uparrow_k \{l_i : \gamma_i, o_p : \gamma'_p\} \\
=&\{l_i : (\alpha_i \uparrow_{k-1} \beta_i) \uparrow_{k-1} \gamma_i\}
\end{aligned}$$

while,

$$\begin{aligned}
&\{l_i : \alpha_i, m_j : \alpha'_j\} \uparrow_k (\{l_i : \beta_i, n_k : \beta'_k\} \uparrow_k \{l_i : \gamma_i, o_p : \gamma'_p\}) \qquad ({}^\forall j, {}^\forall k, {}^\forall p. m_j \neq n_k \neq o_p) \\
=&\{l_i : \alpha_i, m_j : \alpha'_j\} \uparrow_k \{l_i : \beta_i \uparrow_{k-1} \gamma_i\} \\
=&\{l_i : \alpha_i \uparrow_{k-1} (\beta_i \uparrow_{k-1} \gamma_i)\}
\end{aligned}$$

By induction hypothesis, $\{l_i : (\alpha_i \uparrow_{k-1} \beta_i) \uparrow_{k-1} \gamma_i\} = \{l_i : \alpha_i \uparrow_{k-1} (\beta_i \uparrow_{k-1} \gamma_i)\}$.

Case of sum. Omitted.

∎

By definition, $(\alpha \uparrow \beta) \uparrow \gamma = \lim_{k\to\infty}(\alpha \uparrow_k \beta) \uparrow_k \gamma = \lim_{k\to\infty} \alpha \uparrow_k (\beta \uparrow_k \gamma) = \alpha \uparrow (\beta \uparrow \gamma)$. We have defined $\bot \uparrow \alpha = \alpha \uparrow \bot = \bot$ for any type α so that $\alpha \uparrow (\beta \uparrow \gamma) = (\alpha \uparrow \beta) \uparrow \gamma$ for any types α, β and γ. ∎

The next proposition is necessary to prove the completeness of our inference system.

Proposition 3 (Completeness of lub and glb). *Let α and β be ground types. $\gamma \leq \alpha$ and $\gamma \leq \beta$ iff $\gamma \leq \alpha \downarrow \beta$ and $\alpha \leq \gamma$ and $\beta \leq \gamma$ iff $\alpha \uparrow \beta \leq \gamma$ for any type γ.*

Proof. As above, we assume that lubs and glbs are not $\bot$ and primitive types have this property.

($\Longleftarrow$) We show $\alpha \downarrow \beta \leq \alpha$ and $\alpha \leq \alpha \uparrow \beta$ for any types α and β by confirming that $(\alpha \downarrow \beta) *_k \alpha$ and $\alpha *_k (\alpha \uparrow \beta)$ are true for each type constructor.

Case of arrows.

$$\begin{aligned}
((\alpha_1 \to \alpha_2) \downarrow (\beta_1 \to \beta_2)) *_k (\alpha_1 \to \alpha_2) &= ((\alpha_1 \uparrow \beta_1) \to (\alpha_2 \downarrow \beta_2)) *_k (\alpha_1 \to \alpha_2) \\
&= (\alpha_1 *_{k-1} (\alpha_1 \uparrow \beta_1)) \to ((\alpha_2 \downarrow \beta_2) *_{k-1} \alpha_2)
\end{aligned}$$

Case of product.

$$\begin{aligned}
&(\{l_i : \alpha_i, m_j : \alpha'_j\} \downarrow \{l_i : \beta_i, n_k : \beta'_k\}) *_k \{l_i : \alpha_i, m_j : \alpha'_j\} \quad ({}^\forall j, {}^\forall k. m_j \neq n_k) \\
=&\{l_i : \alpha_i \downarrow \beta_i, m_j : \alpha'_j, n_k : \beta'_k\} *_k \{l_i : \alpha_i, m_j : \alpha'_j\} \\
=&\{l_i : (\alpha_i \downarrow \beta_i) *_{k-1} \alpha_i, m_j : \alpha'_j \downarrow \alpha'_j\} \\
=&\{l_i : (\alpha_i \downarrow \beta_i) *_{k-1} \alpha_i, m_j : \alpha'_j\}
\end{aligned}$$

Case of sum.

$$\begin{aligned}
&([l_i : \alpha_i, m_j : \alpha'_j] \downarrow [l_i : \beta_i, n_k : \beta'_k]) *_k [l_i : \alpha_i, m_j : \alpha'_j] \quad ({}^\forall j, {}^\forall k. m_j \neq n_k) \\
=&[l_i : \alpha_i \downarrow \beta_i] *_k [l_i : \alpha_i, m_j : \alpha'_j] \\
=&[l_i : (\alpha_i \downarrow \beta_i) *_{k-1} \alpha_i]
\end{aligned}$$

By simultaneous induction, we have $\alpha \downarrow \beta \leq \alpha$ and $\alpha \leq \alpha \uparrow \beta$ so that $\gamma \leq \alpha \downarrow \beta \leq \alpha$ and $\alpha \leq \alpha \uparrow \beta \leq \gamma$.
($\Longrightarrow$) Consider $\gamma * (\alpha \times \beta)$. Without loss of generality, let $\langle \gamma_i, \langle \alpha_i, \beta_i \rangle \rangle$ be components of $\gamma * (\alpha \times \beta)$ in the positive positions, and $\langle \langle \alpha_i, \beta_i \rangle, \gamma_i \rangle$ be components in the negative positions. We define new operations $\uparrow'$ and $\downarrow'$ such that if $\alpha_i \neq 0$ and $\beta_i \neq 0$, then $\uparrow' \langle \alpha_i, \beta_i \rangle = \alpha_i \uparrow \beta_i$, otherwise $\uparrow' \langle \alpha, 0 \rangle = \uparrow' \langle 0, \alpha \rangle = \alpha$ and $\downarrow'$ is defined similarly. For all components $\langle \gamma_i, \langle \alpha_i, \beta_i \rangle \rangle$, we construct a set of constraints $\gamma_i \leq \downarrow' \langle \alpha_i, \beta_i \rangle$ and for all components $\langle \langle \alpha_i, \beta_i \rangle, \gamma_i \rangle$, we make $\uparrow' \langle \alpha_i, \beta_i \rangle \leq \gamma_i$. Obviously, the set of constraints is equivalent to $\gamma \leq \alpha \downarrow \beta$. Equality $\gamma_i \leq \downarrow' \langle \alpha_i, \beta_i \rangle \Longleftrightarrow \gamma_i \leq \alpha_i$ and $\gamma_i \leq \beta_i$ implies it is also equivalent to $\gamma \leq \alpha$ and $\gamma \leq \beta$.
Therefore, $\gamma \leq \alpha$ and $\gamma \leq \beta$ implies $\gamma \leq \alpha \downarrow \beta$ and vice versa. ▌

Note that the upper bound of a type is the *glb* of a set of the upper bound types.

3. Semantics of the Target Language

The target language is a simple ML-like functional language with product and sum to concentrate our attention to type inference algorithms. This is the language in [Cardelli,1984] added **let**. It has no restrictions such as type declarations or any kind of annotation for type inference. Its extension to imperative languages will be discussed in a later section. The syntax is as follows:

$$\begin{aligned} e \quad ::= \quad & 0 \mid x \mid \textbf{let } x = e \textbf{ in } e \mid \textbf{fix } f(x) = \lambda x.e \textbf{ in } e \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \mid e + e \\ & \textbf{fn } x \Rightarrow e \mid e\,e \mid \{l : e, \cdots, l : e\} \mid e.l \mid l\,e \mid \textbf{case } e \textbf{ of } lx \Rightarrow e \mid \cdots \mid lx \Rightarrow e \end{aligned}$$

where x is a variable and l is a label. The semantics of this language is described in figure I. The semantic function is $\mathcal{E}[\![\ ]\!] : \textbf{Exp} \rightarrow \textbf{Env} \rightarrow \textbf{V}$ where **Exp** are syntactic expressions, **Env** are environments, and **V** are values. **F** denotes the domain of continuous functions $\textbf{V} \rightarrow \textbf{V}$. We may use new constants or primitive functions that do not appear in this table if necessary. The domains of product and sum are coalesced so that

---

core.

$\mathcal{E}[\![0]\!]\rho = 0$ (constant)
$\mathcal{E}[\![x]\!]\rho = \rho x$ (variable)
$\mathcal{E}[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!]\rho = \textit{if } \mathcal{E}[\![e_1]\!]\rho \neq \bot \textit{ then } \mathcal{E}[\![e_2]\!]\rho\{\mathcal{E}[\![e_1]\!]\rho/x\} \textit{ else } \bot$ (let)
$\mathcal{E}[\![\textbf{fix } f(x) = \lambda x.e_1 \textbf{ in } e_2]\!]\rho = \mathcal{E}[\![e_2]\!]\rho\{Y(\lambda v.\mathcal{E}[\![\lambda x.e_1]\!]\rho\{v/f\})/f\}$ (fix)
$\mathcal{E}[\![\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2]\!]\rho = \textit{if } \mathcal{E}[\![e]\!]\rho = \text{true } \textit{then } \mathcal{E}[\![e_1]\!]\rho \textit{ else if } \mathcal{E}[\![e]\!]\rho = \text{false } \textit{then } \mathcal{E}[\![e_2]\!]\rho \textit{ else } \bot$ (if)
$\mathcal{E}[\![e_1 + e_2]\!]\rho = \textit{if } \mathcal{E}[\![e_1]\!]\rho \in \textbf{Int } \textit{and } \mathcal{E}[\![e_2]\!]\rho \in \textbf{Int } \textit{then } \mathcal{E}[\![e_1]\!]\rho + \mathcal{E}[\![e_2]\!]\rho \textit{ else } \bot$ (a primitive function)

function.

$\mathcal{E}[\![\textbf{fn } x \Rightarrow e]\!]\rho = (\lambda v.\mathcal{E}[\![e]\!]\rho\{v/x\})$ (abstraction)
$\mathcal{E}[\![e_1 e_2]\!]\rho = \textit{if } \mathcal{E}[\![e_1]\!]\rho \neq \bot \textit{ and } \mathcal{E}[\![e_2]\!]\rho \neq \bot \textit{ then } (\mathcal{E}[\![e_1]\!]\rho|\mathbf{F})(\mathcal{E}[\![e_2]\!]\rho) \textit{ else } \bot$ (application)

product.

$\mathcal{E}[\![\{l_1 : e_1, \cdots, l_n : e_n\}]\!]\rho = \{l_1 : \mathcal{E}[\![e_1]\!]\rho, \cdots, l_n : \mathcal{E}[\![e_n]\!]\rho\}$ (record construction)
$\mathcal{E}[\![e.l]\!]\rho = \textit{if } \mathcal{E}[\![e]\!]\rho \textit{ has a field } l \textit{ then } (\mathcal{E}[\![e]\!]\rho).l \textit{ else } \bot$ (field selection)

sum.

$\mathcal{E}[\![\textbf{nil}]\!]\rho = [\textbf{nil} : \bot]$ (single constructor)
$\mathcal{E}[\![l\,e]\!]\rho = [l : \mathcal{E}[\![e]\!]\rho]$ (carrier constructor)
$\mathcal{E}[\![\textbf{case } e \textbf{ of } l_1 x_1 \Rightarrow e_1 \mid \cdots \mid l_n x_n \Rightarrow e_n]\!]\rho$
$= \textit{if } \mathcal{E}[\![e]\!]\rho \textit{ matches } [l_i : e'] \textit{ then } \mathcal{E}[\![e_i]\!]\rho\{\mathcal{E}[\![e']\!]\rho/x_i\} \textit{ else } \bot$ (pattern match)

Figure I. Semantics of the Target Language

---

if at least one element is $\bot$, then the whole value is also $\bot$. $|\mathbf{F}$ means a restriction to function space. Y is the fixed-point operator. We assume that identical labels do not appear in a record or a case-statement more than once. The semantics of function application is considered as the applicative order evaluation.

4. Inference Algorithm

In this section, we explain our type inference algorithm and give the proofs of its soundness and completeness. It turns out to be possible to combine subtype analysis and parametric polymorphism naturally. An overview

of the whole algorithm is as follows. First, the constraint graph is generated from a given program. Then, we compute all lower bounds and upper bounds for all free type variables in the graph. Finally, we examine if contradiction exists or not and provide a type assignment for each expression in the program or report errors.

CONST	$A \vdash e : \alpha \quad$ if $e : \alpha \in A$
VAR	$A \vdash x : \alpha[\beta_i/\alpha_i] \quad x : \alpha \in A, \alpha_i$ are the quantified variables of α and β_i are new variables
LET	$\dfrac{A \vdash e_1 : \beta \quad A + \{x : {}^{\forall}\vec{\alpha}.\beta\} \vdash e_2 : \gamma}{A \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \gamma}$
FIX	$\dfrac{A + \{x : \beta, f : \beta \to \gamma\} \vdash e_1 : \gamma \quad A + \{f : {}^{\forall}\vec{\alpha}.\beta \to \gamma)\} \vdash e_2 : \delta}{A \vdash \mathbf{fix}\ f(x) = \lambda x.e_1\ \mathbf{in}\ e_2 : \delta}$
IF	$\dfrac{A \vdash e_1 : \alpha \quad \alpha \leq \mathbf{Bool} \quad A \vdash e_2 : \beta \quad A \vdash e_3 : \gamma \quad \delta = \mathrm{lub}(\beta, \gamma)}{A \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \delta}$
PLUS	$\dfrac{A \vdash e_1 : \alpha \quad A \vdash e_2 : \beta \quad \alpha \leq \mathbf{Num} \quad \beta \leq \mathbf{Num} \quad \gamma = \mathrm{lub}(\alpha, \beta)}{A \vdash e_1 + e_2 : \gamma}$
ABS	$\dfrac{A + \{x : \alpha\} \vdash e : \beta}{A \vdash \mathbf{fn}\ x \Rightarrow e : \alpha \to \beta}$
APP	$\dfrac{A \vdash e_1 : \alpha \to \beta \quad A \vdash e_2 : \alpha' \quad \alpha' \leq \alpha}{A \vdash (e_1 e_2) : \beta}$
RECORD	$\dfrac{A \vdash e_1 : \alpha_1 \quad \ldots \quad A \vdash e_n : \alpha_n}{A \vdash \{l_1 : e_1, \cdots, l_n : e_n\} : \{l_1 : \alpha_1, \cdots, l_n : \alpha_n\}}$
DOT	$\dfrac{A \vdash e : \beta \quad \beta \leq \{l : \alpha\}}{A \vdash e.l : \alpha}$
INJECT	$\dfrac{A \vdash e : \alpha}{A \vdash l\,e : [l : \alpha]}$
MATCH	$\dfrac{A \vdash e : \alpha \quad A + \{x_i : \beta_i\} \vdash e_i : \gamma_i \ (1 \leq {}^{\forall}i \leq n) \quad \alpha \leq [l_1 : \beta_1, \cdots, l_n : \beta_n] \quad \delta = \mathrm{lub}(\gamma_1, \cdots, \gamma_n)}{A \vdash \mathbf{case}\ e\ \mathbf{of}\ l_1\, x_1 \Rightarrow e_1 \mid \ldots \mid l_n\, x_n \Rightarrow e_n : \delta}$

Figure II. Rules for Type Inference

4.1. Generating Subtype Constraint Graph. We infer exact types and generate subtype constraints simultaneously. An exact inference part is similar to ML's system except that product and sum type are handled. The rules are listed in figure II. Concerning the rule LET and FIX, ${}^{\forall}\vec{\alpha}.\beta$ means that all free variables in β except contained in the assumption A are universally quantified.

Several rules can be also looked up at inference rules of partial types, which is proposed in [Thatte,1988]. Partial types aimed to incorporate heterogeneous objects into a ML-like type system and to eliminate run-time type errors. Their heterogeneous objects are in our terminology different type constructors with Ω and safety analysis means subtype inference. This is the reason similar rules appear.

During inference, circular subtype constraint may appear. Instances of circular constraints are $\alpha \leq \cdots \leq \alpha$ or $\alpha \leq \cdots \leq C(\alpha)$ for type variable α and some type constructor C. F-bounded quantification [Canning et al.,1989] is a circular constraint having a form of one inequality. If a set of subtype constraints has no circular constraints, the meaning is clear. But it can not cope with recursive data structures and recursive functions such as [Stansifer,1988]. Therefore, *we solve circular constraints and transform them into equivalent non-circular constraints.* It turns out providing the semantics of F-bounded polymorphism. It will be discussed, however, in the following subsection. First, we obtain a set of constraints through inference rules as follows:

1. A subtype relation between primitive types can be judged immediately.
2. If there is an inequality whose both sides have the same kind of type constructor, we can decompose it into finer constraints.

3. In case of a circular constraint like $\alpha \leq \cdots \leq \alpha$, we consider inequal signs between them as equalities and unify all variables.
4. In case of a circular constraint over a type constructor like $\alpha \leq \cdots \leq C(\cdots\alpha\cdots)$, we exclude circularity.

We can omit completion operations of a constraint graph such as described in [Kozen et al.,1992] since we always have the finest constraints during inference.

After getting subtype constraints, we compute upper bounds and lower bounds for all type variables. Resulting constraints are $\underline{\alpha} \leq \alpha \leq \overline{\alpha}$ where α is a type variable and $\underline{\alpha}, \overline{\alpha}$ are its lower bound and upper bound, respectively. If an upper bound type or a lower bound type contain type variables, it means that dynamic type checking is needed. For instance, consider application function $\mathbf{app} = \mathbf{fn}\ f \Rightarrow \mathbf{fn}\ x \Rightarrow (f\,x)$, whose type is $^{\forall}\alpha.^{\forall}\beta.^{\forall}\alpha'.(\alpha \to \beta) \to \alpha' \to \beta$ and constraint is $\alpha' \leq \alpha$. We can not judge whether this constraint is satisfied or not until all arguments are applied to this function. Of course, if the whole program is given, we can omit dynamic type checking. If we find a variable α such that $\underline{\alpha} > \overline{\alpha}$ or fails to compute lubs or glbs, the solution which satisfies the constraint graph does not exist.

A free type variable which is not under any subtype constraint is used for parametric polymorphism. A type of a function $\mathbf{id} = \mathbf{fn}\ x \Rightarrow x$ is $^{\forall}\alpha.\alpha \to \alpha$ and it has no subtype constraints. This function is completely compatible in Damas and Milner's type system, while constrained functions obey subtyping rules.

A subtype constraint $\leq$ is asymmetric. A lower bound means that the variable may be substituted by an instance of the type while an upper bound means that if a value whose type is greater than the upper bound comes, the corresponding operator fails to receive the value. A lower bound is a concrete value and an upper bound is a constraint of some operation. Hence, we conclude that the type of an expression to be its lower bound type. Consider the following example.

Example 9. $(\mathbf{fn}\ x \Rightarrow x)3$

Suppose 3 is **Int**, then a type of this expression is α and the constraint is $\mathbf{Int} \leq \alpha$. We conclude that the type of this expression is **Int**. But the identification of a type with its lower bound type is only for expressions of types and the constraint $\mathbf{Int} \leq \alpha$ still remains.

4.2. Solving a Circular Constraint. A basic idea for solving circular constraints was provided in [Kaes,1992]. We have proved that it is sufficient. The meaning of F-bounded quantification becomes clear as a result.

Definition 2. A type constructor C over a domain of infinite ground types is a **covariant constructor** if $\alpha \leq \beta$ implies $C(\alpha) \leq C(\beta)$ and if $\alpha > \beta$ holds or α does not have a subtype relation with β(we shall denote it by $\alpha \neq \beta$), then $C(\alpha) \not\leq C(\beta)$.

Note that $C(\alpha) \leq C(\beta)$ implies $\alpha \leq \beta$ since whenever $\alpha > \beta$ or $\alpha \neq \beta$ hold, then $C(\alpha) \not\leq C(\beta)$. Both product and sum are covariant constructors and function is a covariant constructor for its codomain.

Theorem 1. *Let C be a covariant constructor. A type constraint $\alpha \leq C(\alpha)$ is equivalent to $\alpha \leq \mu\tau.C(\tau)$.*

Proof. ($\Longrightarrow$) Suppose that $\alpha \leq C(\alpha)$. Since C is a covariant constructor, $C(\alpha) \leq C(C(\alpha))$. Therefore, by induction, $\alpha \leq C^n(\alpha)$ for each $n \in \mathbf{N}$. C is a contracting mapping over infinite trees [Courcelle,1983] so that $\lim_{n\to\infty} C^n(\alpha)$ exists uniquely and $\lim_{n\to\infty} C^n(\alpha) = \mu\tau.C(\tau)$. By the definition of subtype relation, we have $\alpha \leq \mu\tau.C(\tau)$.
($\Longleftarrow$) Suppose that $\alpha \leq \mu\tau.C(\tau)$ and $\alpha \not\leq C(\alpha)$. Since C is a covariant constructor, $\alpha \leq \mu\tau.C(\tau) \Longrightarrow C(\alpha) \leq C(\mu\tau.C(\tau)) \Longrightarrow C(\alpha) \leq \mu\tau.C(\tau)$. By induction, we have $C^n(\alpha) \leq \mu\tau.C(\tau)$ for all $n \in \mathbf{N}$. If we assume that there is $n \in \mathbf{N}$ for which $\alpha \leq C^n(\alpha)$, then $\alpha \leq C^n(\alpha) \leq \mu\tau.C(\tau)$. Therefore, there is no such n. However, $\alpha *_k \mu\tau.C(\tau)$ is true for all $k \in \mathbf{N}$. We have a contradiction. ∎

A contravariant constructor is defined similarly.

Definition 3. A type constructor C over a domain of infinite ground types is a **contravariant constructor** if $\alpha \leq \beta$ implies $C(\alpha) \geq C(\beta)$ and if $\alpha > \beta$ or $\alpha \neq \beta$ hold, then $C(\alpha) \not\geq C(\beta)$.

A function is a contravariant constructor for its domain.

Theorem 2. *Let C be a contravariant constructor. A type constraint $\alpha \leq C(\alpha)$ is equivalent to $\alpha \leq \mu\tau.C(\tau)$.*

Proof. In this case, the proof is a little complicated. Consider both an ascending sequence and a descending one for which:

1. $\alpha \leq C(C(\alpha)) \leq C^4(\alpha) \leq \cdots$,
2. $C(\alpha) \geq C(C(C(\alpha))) \geq C^5(\alpha) \geq \cdots$.

Note that $C^{2n}(\alpha) \leq C^{2n+1}(\alpha)$ and $C^{2n+1}(\alpha) \geq C^{2n+2}(\alpha)$ hold so that both sequences are bounded. They must converge because C^2 is contractive although they may not converge to the same limit. We denote by $C^{2\infty}$ and $C^{2\infty+1}$ the limits of these two sequences, respectively. In this case, We claim that $\mu\tau.C(\tau)$ has the universal property such that for any sequences, $C^{2\infty} \leq \mu\tau.C(\tau)$ and $\mu\tau.C(\tau) \leq C^{2\infty+1}$ because if we consider only odd elements, the result of the previous theorem can be applied and so is even elements. ▌

Canning writes in [Canning et al.,1989].

> two types t_1 and t_2 may satisfy an F-bound ($t_1 \subseteq F[t_1]$ and $t_2 \subseteq F[t_2]$) but not be in a subtype relation(neither $t_1 \subseteq t_2$ or $t_2 \subseteq t_1$). This means that a F-bounded function may be applied to (or "inherited" by) objects with incomparable types, demonstrating that the inheritance hierarchy is distinct from the subtype hierarchy[Snyder,1986].

where $\subseteq$ means a subtype relation in their terminology. This assertion seems strange. They claim that there are two types t_1 and t_2 which are not in a subtype relation but satisfy $t_1 \leq C(t_1)$ and $t_2 \leq C(t_2)$ for some type constructor C so that F-bounded quantification is not a subtype relation. But from our point of view, their example seems similar to the following one. Consider two products $\{l_1 : \mathbf{Int}, l_2 : \mathbf{Nat}\}$ and $\{l_1 : \mathbf{Int}, l_3 : \mathbf{Str}\}$. They are not in a subtype relation although both are subtypes of $\{l_1 : \mathbf{Int}\}$. As we have proved, F-bounded quantification is equivalent to a subtype constraint if types are restricted to regular types and type variables are treated as Damas and Milner's system.

4.3. Soundness and Completeness. In this subsection, we sketch proofs of soundness and completeness. P denotes a given program and G denotes a constraint graph genereted from P.

Theorem 3 (Soundness). *Let τ be an inferred type(lower bound) of any expression e in P and σ be its upper bound type. If $\tau, \sigma \neq \bot$ and $\tau \leq \sigma$, then e causes no type errors.*

Proof. We assume that if all subtype constraints in every inference rule hold, the program causes no type errors. For instance, in the rule RECORD, if $\beta \leq \{l : \alpha\}$, then the field selection must succeed. Obviously, our inference algorithm preserves these constraints so that $\tau, \sigma \neq \bot$ and $\tau \leq \sigma$ implies that e is one of solutions of G. Therefore, e causes no type errors. ▌

Theorem 4 (Completeness). *For any type constraint graph A and any expression e, if A is consistent with P, then there is a type assignment S such that $S\tau_G \leq \tau_A \leq \sigma_A \leq S\sigma_G$ where τ_G, σ_G are lower and upper bound type of e under G respectively and τ_A, σ_A are under A.*

Proof. From Proposition 3, if $\tau_A, \sigma_A < S\tau_G$ or $S\sigma_G < \tau_A, \sigma_A$, it implies that A is not consistent with P. ▌

Proposition 4 (Uniqueness of principle type constraint). *The resulting subtype constraint is unique.*

Proof. An immediate consequence of Proposition 2. ▌

4.4. Discussions. We think object-oriented languages do not agree with functional properties because an object is a thing that may have side-effects. We consider that one of natural extensions of our inference system to imperative languages is as follows. Each occurrence of the same variable has different type variables. Let α_x and β_x be type variables of the same variable x at different occurrences. If there is a data flow from α_x to β_x, then a type constraint $\alpha_x \leq \beta_x$ is added to the system. Consider the following piece of a program à la C.

$$\begin{aligned} &\textbf{if } (t == 10) \\ &\quad x = 3; \\ &y = x; \end{aligned}$$

Suppose that the type variable of the first occurrence of x is α_x and that of the second occurrence β_x. A data flow from α_x to β_x may exist (if t is equal to 10) so that $\alpha_x \leq \beta_x$ is added to the system.

We restrict types to regular types and general infinite types are not allowed. But all types that are generated by our inference system are regular except the restriction of quantification. Types generated by inference rules are a primitive type, generated by a type constructor or a solution of a subtype constraint. A primitive type is obviously regular. A length of a program is finite so that type construction always terminates in finite steps. Hence, if all components are regular, the result is also regular. Lub and glb operations generate regular types from regular types. Consequently, all types are regular.

The restriction of quantification does not approve records having the following type:

$$\mu\tau.\{l : {}^{\forall}\alpha.\alpha \to \alpha, m : \tau\}$$

An instance of the type is:

$$\{l : \mathbf{Int} \to \mathbf{Int}, m : \{l = \mathbf{Nat} \to \mathbf{Nat}, m : \{l = \mathbf{Str} \to \mathbf{Str}, \cdots\}\}\}$$

In order to cope with such a type, we need matching and unification of non-regular infinite trees as pointed out in [Kaes,1992].

5. Examples

Consider the following ML-like function which computes sum of numbers in a list:

$$\begin{aligned}&\textbf{fun SumList nil} = 0\\ &\quad \mid \textbf{SumList List}(\textbf{value}, \textbf{rest}) = \textbf{value} + \textbf{SumList}(\textbf{rest})\end{aligned}$$

This function is translated into our language as follows:

$$\textbf{fix SumList}(x) = \lambda x.\textbf{case } x \textbf{ of nil} \Rightarrow 0 \mid \textbf{List}(v, r) \Rightarrow v + \textbf{SumList}(r) \textbf{ in } \cdots$$

Its type is inferred as follows. In the first step, by the rule FIX,

$$\frac{\{x : \alpha, \textbf{SumList} : \alpha \rightarrow \beta\} \vdash \textbf{case } x \textbf{ of nil} \Rightarrow 0 : \textbf{Int} \mid \textbf{List}(v, r) \Rightarrow v + \textbf{SumList}(r) : \varepsilon}{\{\} \vdash \textbf{fix SumList}(x) = \lambda x.\textbf{case } x \textbf{ of nil} \Rightarrow 0 \mid \textbf{List}(v, r) \Rightarrow v + \textbf{SumList}(r) \textbf{ in } \cdots} \text{ FIX}$$

Next, the latter half of the case statement is inferred as follows:

$$\frac{v : \gamma \quad r : \delta \quad \gamma \leq \textbf{Num} \quad \beta \leq \textbf{Num} \quad \delta \leq \alpha \quad \varepsilon = \text{lub}(\gamma, \beta)}{\{x : \alpha, \textbf{SumList} : \alpha \rightarrow \beta\} \vdash v + \textbf{SumList}(r) : \varepsilon} \text{ PLUS}$$

The whole case statement is inferred by MATCH.

$$\frac{\alpha \leq [\textbf{nil} : \bot, \textbf{List} : \gamma \times \delta] \quad \beta = \text{lub}(\textbf{Int}, \varepsilon)}{\{x : \alpha, \textbf{SumList} : \alpha \rightarrow \beta\} \vdash \textbf{case } x \textbf{ of nil} \Rightarrow 0 \mid \textbf{List}(v, r) \Rightarrow v + \textbf{SumList}(r) : \varepsilon} \text{ MATCH}$$

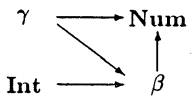
Some trivial deductions (for instance, $0 : \textbf{Int}$) are omitted. Then, we get a set of constraints listed as follows: ($\alpha = \text{lub}(\beta, \gamma)$ is interpreted as $\beta \leq \alpha$ and $\gamma \leq \alpha$.)

$$\begin{aligned}&\alpha \leq [\textbf{nil} : \bot, \textbf{List} : \gamma \times \delta]\\ &\gamma \leq \textbf{Num}, \beta \leq \textbf{Num}, \delta \leq \alpha\\ &\gamma \leq \varepsilon, \beta \leq \varepsilon\\ &\textbf{Int} \leq \beta, \varepsilon \leq \beta\end{aligned}$$

α and δ occurs circularly. Solving the circularity, we have $\alpha \leq \mu\tau.[\textbf{nil} : \bot, \textbf{List} : \gamma \times \tau]$. β and ε occur circularly, too. Hence, they are unified. Consequently, we have

$$\alpha \leq \mu\tau.[\textbf{nil} : \bot, \textbf{List} : \gamma \times \tau], \gamma \leq \textbf{Num}$$

The rest of the constraints is graphically represented as:



This graph has no contradiction, i.e., has solutions. The constraints are minimum so that they themselves are constraints of **SumList** that imposes the outer program. We conclude that **SumList** has the lower bound type, $\mu\tau.[\textbf{nil} : \bot, \textbf{List} : \textbf{Num} \times \tau] \rightarrow \textbf{Num}$.

SumList can accept all instance whose type is less than or equal to $\mu\tau.[\textbf{nil} : \bot, \textbf{List} : \textbf{Num} \times \tau]$. Suppose that $\textbf{Int} \leq \textbf{Real} \leq \textbf{Num}$. For instance,

nil	**(Int)**
[10, 20, 30, 40]	**(Int)**
[0.32, 5.5, 100]	**(Real)**

where () is a type of the return value.

6. Subtyping and Subclassing

Up to now, polymorphism of object-oriented languages is based on subclassing. The reasons why this has worked well are (i). the fact that a record type having more labels is a subtype of one having fewer labels goes with the way to inherit super class, that is making a subclass by adding new labels. (ii). the object-oriented language such as C++ has very strong restriction on inheritance. It makes all virtual functions in inheritance relation having the same type so that inheritance anomaly never occurs. However, as described in [Cook et al.,1990], subclassing and subtyping are distinct notions and subclassing-based polymorphism has possibility to bring about dynamic type errors. Subtyping is ordered structure obtained by abstracting a concept of "type safeness". As shown by this study, it is easy to extract subtype constraints from a program. Meanwhile, we consider that subclassing does not have such a fruitful structure because inheritance hierarchy is an arbitrary partially ordered set *given by a programmer*. The reason there are a few profound theories on subclassing-based inheritance is that it has to deal with an arbitrary partially ordered set.

We consider that inheritance is a tool only for incremental programming and that subtyping is used for a source of polymorphism. Therefore, when we write a subclass, it may not be a subtype of the super class. In this case, an instance of the subclass can not be assigned to a variable of the superclass. For this reason, we think when superclass is inherited, any kind of modification such as addition of a new method, deletion, etc., are allowed in our type system.

7. Conclusion

We have presented a type system and its inference system with recursive data structures in the presence of subtyping by considering a type as a regular tree. Besides, our system keeps several properties that Damas and Milner's system has such as existence of principle types and syntactic completeness of type checking algorithm. From the point of our inference system, subtyping-based polymorphism is considered to be natural. We have provided a semantics of F-bounded polymorphism by defining a type as a regular tree and using the parametric type polymorphism of Damas and Milner's type system. Because our inference system requires a weaker condition, we expect that our system is applicable to practical programming languages widely. Although we have investigated in the framework of a simple functional language, its extension to imperative languages is not difficult. We believe our type system enables ML to incorporate subtyping naturally.

There are several areas that need further investigation. These are (i). to estimate the complexity of our inference algorithm. (ii). to provide semantics of our inference system on a model of recursive types such as an ideal model[MacQueen et al.,1986] and a PER model[Bruce et al.,1992] (iii). implementation of our algorithm on interpreters and compilers. As for (iii), an inference interpreter is currently under development. (iv). to investigate an efficient implementation scheme.

Acknowledgement

We would like to thank Atsushi Ohori who was kind enough to read the earlier version of this paper and gave us helpful suggestions. We would also like to thank Jacques Garrigue for comments.

Bibliography

[Amadio et al.,1991] Robert M. Amadio and Luca Cardelli. Subtyping Recursive Types. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 104–118, January 1991.

[Baumgartner et al.,1992] Gerald Baumgartner and Vincent F. Russo. Type Abstraction and Subtype Polymorphism for Object-Oriented Languages(draft). Technical report, Purdue University, 1992.

[Bruce et al.,1992] Kim Bruce and John C. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 316–327, January 1992.

[Canning et al.,1989] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. F-Bounded Polymorphism for Object-Oriented Programming. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[Cardelli et al.,1985] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. In *Computing Surveys*, volume 17, 1985.

[Cardelli,1984] Luca Cardelli. A Semantics of Multiple Inheritance. In D.B. MacQueen G. Kahn and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984.

[Cook et al.,1990] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance Is Not Subtyping. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, January 1990.

[Courcelle,1983] Bruno Courcelle. Fundamental Properties of Infinite Trees. In *Theoretical Computer Science*, volume 25, pages 95–169, March 1983.

[Damas,1985] Luis Damas. *Type Assignment in Programming Languages.* PhD thesis, Department of Computer Science, University of Edinburgh, April 1985. CST-33-85.

[Kaes,1992] Stefan Kaes. Type Inference in the Presence of Overloading, Subtyping and Recursive Types. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 193–204, 1992.

[Kozen et al.,1992] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient Inference of Partial Types. In *FOCS*, 1992.

[MacQueen et al.,1986] David MacQueen, Gordon Plotkin, and Ravi Sethi. An Ideal Model for Recursive Polymorphic Types. In *Information and Control*, volume 71, pages 95–130, 1986.

[Milner,1978] R. Milner. A Theory of Type Polymorphism in Programming. In *Journal of Computer and System Sciences*, volume 17, 1978.

[Snyder,1986] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 38–45, 1986.

[Stansifer,1988] Ryan Stansifer. Type Inference with Subtypes. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 88–97, January 1988.

[Thatte,1988] Satish Thatte. Type Inference with Partial Types. In *Proceedings of Internatinal Colloquium on Automata, Languages, and Programming*, volume 317 of *Lecture Notes in Computer Science*, pages 615–629. Springer-Verlag, 1988.

[Wand,1987] Mitchell Wand. Complete Type Inference for Simple Objects. In *Proceedings of Second Symposium on Logic in Computer Science*, pages 37–44, 1987.