Electronic Notes in Theoretical Computer Science **14** (1998)
URL: http://www.elsevier.nl/locate/entcs/volume14.html 14 pages

Towards Higher-Order Types

Carlos Camarão

Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, 31270-010 Belo Horizonte, Brasil

Lucília Figueiredo

Departamento de Computação, Universidade Federal de Ouro Preto, 35400-000 Ouro Preto, Brasil

Abstract

This article explores the use of types constrained by the definition of functions of given types. This notion supports both overloading and a form of subtyping, and is related to Haskell type classes and System O. We study an extension of the Damas-Milner system, in which overloaded functions can be defined. The inference system presented uses a context-independent overloading policy, specified by means of a predicate used in a single inference rule. The treatment of overloading is less restrictive than in similar systems. Type annotations are not required, but can be used to simplify inferred types. The work motivates the use of constrained types as parameters of other, higher-order types.

1 Introduction

The problems with the treatment of overloading in languages with support for polymorphism and type inference, such as for example Miranda [11] and SML [6,9], have been discussed elsewhere [13,3]. In SML, for example, one cannot write:

```
square x = x * x
```

the reason coming from the fact that `'*'` is overloaded for integers and reals. Equality is treated differently in SML, by the introduction of a special polymorphic type variable, constrained so that its instances must admit equality. For example, the type of a function `member`, that tests membership in a list, is given as follows:

```
''a list -> ''a -> bool
```

In Miranda, this type is not constrained in this way, but applying `member` to lists whose elements are functions generates a run-time error.

Type classes [2] are used in Haskell [4,10] to deal with problems like these. A type class is introduced to specify types of overloaded functions, as in:

```
class Num a where
     (+), (*)::  a -> a -> a
      ...
```

and:

```
class Eq a where
     (==)::  a -> a -> bool
```

Instance declarations then specify which types are instances of this class, and give definitions of overloaded functions for this type, as in:

```
instance Num Int where
    (+) = PrimAddInt
     ...
```

and:

```
instance Eq Int where
    (==) = PrimEqInt
     ...
```

and also:

```
instance Eq a,b => Eq (a,b) where
    (a,b) == (c,d) = (a==c) & (b==d)
```

The last example illustrates subclassing: if equality is defined on `a` and `b`, then it is defined on the product type `(a,b)`.

System O [8] aimed at some improvements in relation to type classes:

- With type classes, inferred types depend on class declarations. This is in constrast with the Damas-Milner system, for which all type declarations

can be removed from any typeable program and the program still remains typeable.

- Type classes cannot be defined to be formed by polymorphic type instances: for example, one cannot define a type class formed by product types `(a,b)`, `(a,b,c)`, ... , all having, say, a projection function `first`. [1]

System O uses universally quantified types, with possibly a constraint on the quantified variable that is a (possibly empty) set of bindings $o :: \alpha \to \tau$, indicating that there must exist an overloaded operator $o :: p \to (\tau[\alpha :\to p])$, for an instance p of α. [2]

The main difference of our system in relation to System O is that we eliminate some restrictions imposed in System O in the definition and use of overloaded symbols. System O requires explicit declaration of the type of an overloaded function (our system does not). Thus, there is no inference of types of overloaded symbols in System O. In System O, the argument of an overloaded function must be of a type that is constructed from a given type constructor (including $\to$, the constructor of function types), and all the argument type constructors of overloaded functions must be (pairwise) distinct. As an example, functions with the following types cannot be overloaded in System O:

$$\texttt{sort} : (\alpha \to \alpha \to \texttt{Bool}) \to [\alpha] \to [\alpha]$$

and

$$\texttt{sort} : (\alpha \to \texttt{Int}) \to [\alpha] \to [\alpha]$$

We say that a quantified constrained type is a *class type*. A class type may be viewed as representing a collection of (lower-order) types, the set of its *instances*, all having given (overloaded) functions that operate on parameters of corresponding types. These instance types may be concrete and abstract types. (The implications of using class types as the type of abstract types are not dealt with in this paper, being left as a topic for further work.) In a system with type declarations, class types can be used in type annotations to simplify inferred types, as illustrated in Section 2.

The rest of the paper is organized as follows. Section 2 explores some uses of class types. Section 3 introduces the type rules of our system. Section 4 presents the type inference algorithm. Section 5 gives a semantics for terms and type expressions of System CT. Section 6 concludes.

[1] Where `first(a,b) = a, first(a,b,c) = a, ...`

[2] The notation $\sigma[\alpha :\to \tau]$ indicates the usual textual substitution.

2 Examples

In this section we explore some uses of class types, using a Haskell-like notation.

2.1 Types with Equality and Ordering Relations

We consider first the possibility of defining a type that represents any type over which equality (say '==') is defined:

The equality class type is as follows:

```
type Eq t = t { (==):: t -> t -> Bool }
```

Type `Int` is automatically an instance of `Eq t` if there is a built-in operator '==' with the required type. Type `Eq t` can be seen — in a language supporting class types and type definitions of this form — as an abbreviation for $\forall t\,\{\texttt{(==)} : t \to t \to \texttt{Bool}\}.\, t$.

A function `member` to test membership on lists of elements that can be compared on equality can be written as follows:

```
member:: [Eq t] -> t -> Bool
member _ [] = false
member a (b:x) = (a==b) || member a x
```

Using class types, the mechanism corresponding to Haskell instance declarations is not used (it would correspond to an explicit declaration of subtyping).

Types with an ordering relation (say '<') are treated analogously:

```
type Ord t = t { (<): t→ t→ Bool }
```

As with equality, type `Int` is an instance of `Ord t` if there is an operator '<' with type `Int`→`Int`→`Bool`. For any t_1 that is an instance of `Ord t`, type $[t_1]$ becomes (automatically) an instance of `Ord t` by defining:

```
[] < _ = false
_ < [] = true
(a1:x1) < (a2:x2) = if a1<a2 then x1<x2 else false
```

The inferred type of this particular definition of '<' can be `[Ord t]`→`[t]`→ `Bool`. The constraint information (`Ord`) needs to occur only once. The type

`[Ord t]`$\rightarrow$`[t]`$\rightarrow$ `Bool` can be seen as an abbreviation for

$$\forall t\{(<) : t \rightarrow t \rightarrow \texttt{Bool}\}.\, [t] \rightarrow [t] \rightarrow \texttt{Bool}$$

2.2 Arithmetic Operations

For an example of a class type `Num` constraining types with arithmetic operations, consider:

```
type Num t = t { (+), (*)::  t -> t -> t
                 negate::  t -> t ... }
```

A square function can be written as:

```
square::  Num t -> t
square x = x * x
```

The type annotation is used to indicate a more specific type (a subtype), and provides an abbreviated notation, but is not required (see Section 4).

We can then write:

```
squares::  (Num a, Num b, Num c) -> (a, b, c)
squares (x,y,z) = (square x, square y, square z)
```

Finally, note that class types can also be used as (super)types of abstract types, in a type system that allows the use of alternative abstract type implementations for a given abstract type signature, based on the overloading of functions and constants defined in these alternative implementations.

3 Type System

We use a kernel language that is almost identical to Core-ML [5,1]. We include value constructors ($k \in \mathcal{K}$) and type constructors ($C \in \mathcal{C}$) and assume (for simplicity) that overloaded variables are distinct from value constructors and non-overloaded variables. All lambda-bound variables are non-overloaded.

Term variables ($x \in X$) are considered to be divided into three groups: of overloaded ($o \in O$), non-overloaded ($u \in U$), and value constructors ($k \in \mathcal{K}$), the latter being considered as constants, having a value fixed in a global environment. Figure 1 gives the syntax of pre-terms and types of system CT.

Types are modified (with respect to the type system of Core-ML) to include class types. A class type $\forall\alpha\{o_1 : \tau_1, \ldots, o_n : \tau_n\}.\, \sigma$ represents all types $\sigma[\alpha :\rightarrow \tau]$ constrained to have functions with types $o_1 : \tau_1', \ldots, o_n : \tau_n'$, where $\tau_i' \equiv \tau_i[\alpha :\rightarrow \tau]$, for $i = 1, \ldots, n$.

$$\begin{array}{lll}
\text{Terms } e \in E & e ::= x \mid \lambda u.e \mid e\ e' \mid \texttt{let}\ x = e\ \texttt{in}\ e' & \\
\text{Simple Types } \tau \in \mathcal{T} & \tau ::= \alpha \mid \tau \to \tau' \mid C\ \tau_1 \ldots \tau_n & (n \geq 0) \\
\text{Types } \sigma \in T & \sigma ::= \tau \mid \forall \alpha\{o_1 : \tau_1, \ldots, o_n : \tau_n\}.\ \sigma & (n \geq 0)
\end{array}$$

Fig. 1. Abstract Syntax of system CT

$$\Gamma, x : \sigma \vdash x : \sigma \qquad \text{(VAR)}$$

$$\frac{\Gamma, u : \tau \vdash e : \tau'}{\Gamma \vdash \lambda u.\, e : \tau \to \tau'} \qquad \text{(ABS)}$$

$$\frac{\Gamma \vdash e : \tau' \to \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash e\ e' : \tau} \qquad \text{(APPL)}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma, x : \sigma \vdash e' : \tau}{\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : \tau} \qquad \text{(LET)}$$

$$\frac{\Gamma, o_i : \tau_i \vdash e : \sigma}{\Gamma \vdash e : \forall \alpha\{o_i : \tau_i\}.\ \sigma}\ \ \alpha \notin tv(\Gamma) \qquad \text{(GEN)}$$

$$\frac{\Gamma \vdash e : \forall \alpha\{o_i : \tau_i\}.\sigma \qquad \Gamma \vdash o_i : \tau_i[\alpha :\to \tau]}{\Gamma \vdash e : \sigma[\alpha :\to \tau]} \qquad \text{(INST)}$$

Fig. 2. Type Rules of System CT

The type rules are given in Figure 2. A *typing context* Γ is a set of pairs, written as $x : \sigma$. In our system, a variable x can occur more than once in a typing context, if $x \in O$. A pair $x : \sigma$ is called a *typing for* x. The notation Γ_x indicates a typing context for which it is assumed that x does not appear (this does not cause any restrictions due to the possibility of renaming bound variables).

The overloading policy is controlled by a predicate ρ, used in rule (LET). The value given by $\rho(\sigma_1, \sigma_2)$ is true if "σ_1 and σ_2 can be types of overloaded function symbols". The evaluation of $\rho(\sigma_1, \sigma_2)$ basically tests if σ_1 and σ_2 are not unifiable; for a context-independent overloading policy the following additional conditions must hold:

- the unification of argument types of functions must fail, and
- if the argument types are functional types, they must be such that they cannot be the types of overloaded functions (in other words, unification of the argument types of the argument types must not fail).

Predicate ρ is defined by:

$$\rho(\sigma_1,\sigma_2) = \begin{cases} unify(\{\tau_1 = \tau_2\}) \text{ fails and} & \\ \text{if } \tau_1 = \tau_{1a} \to \tau_{1r} \text{ and} & \text{if } \sigma_1 = \tau_1 \to \tau_1', \\ \quad \tau_2 = \tau_{2a} \to \tau_{2r}, \text{ then} & \sigma_2 = \tau_2 \to \tau_2' \\ unify(\{\tau_{1a} = \tau_{2a}\}) \text{ does not fail} & \\ \rho(\sigma_1', \sigma_2) & \text{if } \sigma_1 = \forall\alpha\{o_i : \tau_i\}.\, \sigma_1', \\ & \quad \sigma_2 \in \mathcal{T} \\ \rho(\sigma_2', \sigma_1) & \text{if } \sigma_2 = \forall\alpha\{o_i : \tau_i\}.\, \sigma_2', \\ & \quad \sigma_1 \in \mathcal{T} \\ \rho(\sigma_1', \sigma_2') & \text{if } \sigma_1 = \forall\alpha\{o_i : \tau_i\}.\, \sigma_1', \\ & \quad \sigma_2 = \forall\alpha'\{o_j' : \tau_j'\}.\, \sigma_2' \end{cases}$$

Quantified type variables in a given type are assumed to be distinct from other type variables (possibly by renaming).

We also use a function $tc : T \to TC$, that gives the constructor of a type given as parameter, defined below, where we let $TC = \mathcal{C} \cup \{\to, *\}$:

$$\begin{aligned} tc(\forall\alpha\{o_i : \tau_i\}.\, \sigma) &= tc(\sigma) \\ tc(\alpha) &= * \\ tc(\tau \to \tau') &= \to \\ tc(C\ \tau_1 \ldots \tau_n) &= C \end{aligned}$$

We use the notation $\Gamma, x : \sigma$ to stand for:

$$\Gamma, x:\sigma = \begin{cases} \Gamma_x \cup \{x:\sigma\} \text{ if } x \in U \\ \Gamma \cup \{x:\sigma\} \quad \text{if } x \in O \text{ and } tc(\sigma) = \rightarrow \text{ and} \\ \qquad\qquad \{x:\sigma'\} \in \Gamma \Rightarrow \rho(\sigma, \sigma') \end{cases}$$

The condition "$tc(\sigma) = \rightarrow$", which implies that only functions may be overloaded, simplifies the semantics and implementation of the system, since it enables a direct unique identification of the function to be called, in a context-independent way.

We use: $\Gamma, (x_i : \sigma_i)_{i \in \{1,\ldots,n\}}$ as an abbreviation for $\Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n$ and assume systematically that i ranges from 1 to n, where $n \geq 0$, to write only $\Gamma, x_i : \tau_i$; analogously, $\Gamma \vdash (x_i : \sigma_i)$ is used as an abbreviation for $\Gamma \vdash x_1 : \sigma_1 \ \ldots \ \Gamma \vdash x_n : \sigma_n$, and j is assumed systematically to range from 1 to m, where $m \geq 0$, to write $\forall \alpha_j \{o_i : \tau_i\}.\, \sigma$ as an abbreviation for $\forall \alpha_1 . \forall \alpha_2 \ldots . \forall \alpha_m \{o_1 : \tau_1, \ldots o_n : \tau_n\}.\, \sigma$.

The notation $tv(\Gamma)$ stands for the set of free type variables of Γ.

4 Type inference

Figure 3 presents the type inference algorithm. Function PP computes principal pairs (type and context) for a given term, together with a set of constraints for the computed type. We write algorithms using a pattern-matching notation, resembling modern functional programming languages.

Our algorithm PP works only with simple types, for simplicity. Quantified types are equivalent to simple types with a set of constraints $o_i : \tau_i$. We can do this simplification because lambda-bound variables always have simple types and because we can type an expression `let` $x = e$ `in` e' by first finding the possibly polymorphic principal typing for e, and then using instances of this type for each occurrence of x in e'.

For simplicity, we do not consider α-substitutions and assume that if a variable is let-bound, then it is not lambda-bound. We can see that this assumption is important in examples for which the assumptions does not hold, like `let` $x = 10$ `in` $\lambda x.\, x$ *or* $true$, for which PP would fail, and `let` $x = 10$ `in` $\lambda x.\, x$, for which PP would not give a principal typing.

Variable κ is used to denote a set of constraints. We use *typing environments* A, which are sets of pairs written in the form $x : (\tau, \kappa, \Gamma)$, where the second element is a triple (whose first element is a simple type, the second is a set of constraints and the third is a typing context). We write $A(x)$ for the set of triples $(\tau_x, \kappa_x, \Gamma_x)$ such that $x : (\tau_x, \kappa_x, \Gamma_x) \in A$.

A *type substitution* (or simply substitution) is a function from type variables to types. If σ is a type and S is a substitution, then $S\sigma$ is used to denote the type obtained by replacing each free type variable α in σ with $S(\alpha)$. Sim-

ilarly, for a typing context Γ, the notation $S\Gamma$ denotes $\{x : S\sigma \mid \ x : \sigma \in \Gamma\}$, and for a set of constraints κ, the notation $S\kappa$ denotes $\{o : S\tau \mid \ o : \tau \in \kappa\}$.

Functions lcg, $unify$ and sat, also used by algorithm PP, are defined below. Function lcg computes the type that is the *least common generalisation* for a set of types. Function $unify$ computes the most general unifying substitution for a set of equations between type expressions. Function $sat(\kappa, A)$ is a constraint satisfaction function: it returns true if constraints κ are satisfied in typing environment A, and false otherwise. The functions are as follows.

- lcg is defined by:

$$lcg(\{\tau\}) = \tau \quad \text{(i.e. } lcg \text{ of a singleton is the single element)}$$

$$lcg(\mathcal{S} \cup \{C\ \tau_1 \ldots \tau_n,\ C'\ \tau'_1 \ldots \tau'_n\}) =$$

$$\textbf{if } C \not\equiv C' \textbf{ then } \alpha, \text{ where } \alpha \text{ is a fresh type variable}$$

$$\textbf{else } lcg(\mathcal{S} \cup \{C\ lcg_1\ \ldots\ lcg_n\})$$

$$\text{where } lcg_i = lcg(\{\tau_i, \tau'_i\}), \text{ for } i = 1, \ldots, n$$

$$\text{and type variables are renamed so that } \alpha \equiv \alpha' \text{ whenever}$$

$$\text{there are } \tau_a, \tau_b \text{ with } lcg(\{\tau_a, \tau_b\}) = \alpha \text{ and } lcg(\{\tau_a, \tau_b\}) = \alpha'$$

$$lcg(\mathcal{S} \cup \{\alpha\}) = \alpha$$

$$lcg(\mathcal{S} \cup \{\tau_1 \to \tau_2, \tau'_1 \to \tau'_2\}) = \text{ analogous as above,}$$

$$\text{with “}\to\text{” as the type constructor}$$

Function lcg takes into account the fact that, for example, $lcg(\{\texttt{Int} \to \texttt{Int}, \texttt{Bool} \to \texttt{Bool}\})$ is $\alpha \to \alpha$, for some type variable α, and not $\alpha \to \alpha'$, for some other type variable $\alpha' \not\equiv \alpha$.

- $unify$ is given by:

$$unify(\emptyset) = \emptyset$$

$$unify(E \cup \{C\ \tau_1 \ldots \tau_n = C'\ \tau'_1 \ldots \tau'_n\}) =$$

$$\textbf{if } C \not\equiv C' \textbf{ then } \text{fail}$$

$$\textbf{else } unify(E \cup \{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\})$$

$$unify(E \cup \{\alpha = \tau\}) =$$

$$\textbf{if } \alpha \equiv \tau \textbf{ then } unify(E)$$

$$\textbf{else if } \alpha \text{ occurs in } \tau \textbf{ then fail}$$

$$\textbf{else } unify(E[\alpha :\to \tau]) \circ \{\alpha \mapsto \tau\}$$

$$unify(E \cup \{\tau_1 \to \tau_2 = \tau'_1 \to \tau'_2\}) = unify(E \cup \{\tau_1 = \tau'_1, \tau_2 = \tau'_2\})$$

$PP(u, A) =$
  if $A(u) = (\tau, \kappa, \Gamma)$, for some τ, κ, Γ, **then** (τ, κ, Γ)
  else $(\alpha, \emptyset, \{u : \alpha\})$, where α is a fresh type variable

$PP(o, A) =$
  if $A(o) \neq \emptyset$ **then** $\big(\tau,\ \{o : \tau\},\ \Gamma\big)$
    where $A(o) = \{(\tau_i, \kappa_i, \Gamma_i)\},\ \Gamma = \bigcup \Gamma_i$ and $\tau = lcg(\{\tau_i\})$
  else $(\alpha, \emptyset, \{o : \alpha\})$, where α is a fresh type variable

$PP(\lambda u.e, A) =$
  let $PP(e, A) = (\tau', \kappa, \Gamma)$ **in**
    if $u : \tau \in \Gamma$, for some τ
      then $(\tau \to \tau',\ \kappa,\ \Gamma - \{u : \tau\})$
      else $(\alpha \to \tau',\ \kappa,\ \Gamma)$, where α is a fresh type variable

$PP(e\ e', A) =$
  let $PP(e, A) = (\tau, \kappa, \Gamma)$
    $PP(e', A) = (\tau', \kappa', \Gamma')$, with type variables renamed
      to be different from those in (τ, κ, Γ)
    $S = unify(\{\alpha = \alpha' \mid\ x : \alpha \in \Gamma \text{ and } x : \alpha' \in \Gamma'\} \cup \{\tau = \tau' \to \alpha''\})$,
      where α'' is a fresh type variable
  in if $sat(S\kappa, A)$ **then** $\big(S\alpha'', S\kappa \cup S\kappa',\ S\Gamma \cup S\Gamma'\big)$ **else** fail

$PP($`let x=e in e'`$, A) =$
  let $PP(e, A) = (\tau, \kappa, \Gamma)$
    if $x \in O$ and $tc(\tau) = \to$ and $\big(\{x : (\tau', \kappa', \Gamma')\} \in A \Rightarrow \rho(\tau, \tau')\big)$
      then $A' = A \cup \{x : (\tau, \kappa, \Gamma)\}$
      else if $x \in U$
        then $A' = A_x \cup \{x : (\tau, \kappa, \Gamma)\}$
        else $A' = \emptyset$
  in if $A' = \emptyset$ **then** fail **else** $PP(e', A')$

Fig. 3. Type inference for system CT

- $sat(\kappa, A)$ is defined by:
  For each $o : \tau$ in κ, we have: there exists $o : (\tau', \kappa', \Gamma')$ in A such that $unify(\tau = \tau')$ does not fail and, in this case, letting $S = unify(\{\tau = \tau'\})$, $sat(S\kappa', A - \{o : (\tau', \kappa', \Gamma')\})$ also holds.

5 Semantics

Following [7], we use an applicative structure $\mathcal{A}$ that is a tuple:

$$\mathcal{A} = (U_1^{\mathcal{A}}, U_2^{\mathcal{A}}, \mathbf{App}^{\tau,\tau'}, \mathcal{I})$$

where:

- $U_1^{\mathcal{A}} = \{A^\tau\}$ is the collection of sets A^τ constructed inductively as follows:

  - Base case: $\tau \equiv C$ (i.e. $C\ \tau_1 \ldots \tau_n$, where $n = 0$). Let K_τ be the set of all value constructors k that yield a value in τ. Then $A^\tau = \{\mathcal{I}(k) \mid\ k \in K_\tau\}$.
  - Inductive cases:
    $\tau \equiv C\ \tau_1 \ldots \tau_n$, where $n > 0$ and $A^{\tau_1}, \ldots, A^{\tau_n} \in U_1^{\mathcal{A}}$: let K_τ be the set of all value constructors that yield a value in τ. Then, for all $v_i \in A^{\tau_i}$, $i = 1, \ldots, arity(k)$, $A^\tau = \bigcup_{k \in K_c} \mathcal{I}(k)\ v_1 \ldots v_{arity(k)}$.
    $\tau \equiv \tau_1 \to \tau_2$, where $A^{\tau_1}, A^{\tau_2} \in U_1^{\mathcal{A}}$: then $A^\tau = A^{\tau_1} \to A^{\tau_2}$, the set of functions from A^{τ_1} to A^{τ_2}.

- $U_2^{\mathcal{A}} = \{A^\sigma\}$ is the collection of sets A^σ constructed inductively as follows:
  - Base case: $A^\tau \in U_1^{\mathcal{A}}$; then $A^\sigma = A^\tau \in U_2^{\mathcal{A}}$;
  - Inductive case: $A^{\sigma_1}, A^{\sigma_2} \in U_2^{\mathcal{A}}$; then $A^\sigma = A^{\tau_1} \cup A^{\tau_2} \in U_2^{\mathcal{A}}$.
- $\mathbf{App}^{\tau,\tau'}$ is the function

$$\mathbf{App}^{\tau,\tau'} : A^{\tau \to \tau'} \to (A^\tau \to A^{\tau'})$$

  from $A^{\tau \to \tau'}$ to functions from A^τ to $A^{\tau'}$, defined by:

$$\mathbf{App}^{\tau,\tau'}\ f\ x = f(x)$$

- $\mathcal{I} : \mathcal{K} \to \bigcup_{\sigma \in U_2^{\mathcal{A}}} A^\sigma$ assigns values to value constructors.

An environment is a mapping

$$\eta : Variables \to U_1^{\mathcal{A}} \cup \bigcup_{\sigma \in U_2^{\mathcal{A}}} A^\sigma$$

where $Variables$ include (term) variables and type variables, for every type variable α we have $\eta(\alpha) \in U_1^{\mathcal{A}}$, and for every variable x we have $\eta(x) \in A^\sigma$, for some σ.

The meaning $[\![\sigma]\!]\eta$ of a type expression σ in environment η is defined inductively as follows:

$$[\![\alpha]\!]\eta = \eta(\alpha)$$

$$[\![\tau \to \tau']\!]\eta = \{f \mid\ x \in [\![\tau]\!]\eta \Rightarrow f(x) \in [\![\tau']\!]\eta\}$$

$$[\![C\ \tau_1 \ldots \tau_n]\!]\eta = \bigcup\{\mathcal{I}(k)\ v_1 \ldots v_n \mid\ v_i \in [\![\tau_i]\!]\eta,\ \text{for all } i = 1, \ldots, n\}$$

$$[\![\forall\alpha\{o_i : \tau_i\}.\,\sigma]\!]\eta = \bigcap\{[\![\sigma[\alpha :\to \tau]]\!]\eta \mid\ \tau \text{ is such that } \eta(o_i) \in [\![\tau_i[\alpha :\to \tau]]\!]\eta\}$$

The denotation of, for example, $\texttt{Int} \to \texttt{Bool}$ is a set that includes denotations of overloaded functions $f : \sigma$ for which `Int` is an instance of the argument type of σ and `Bool` is an instance of the result type of σ. The intersection used in the denotation of $\forall\alpha\{x_i : \tau_i\}.\,\sigma$ "selects" thus overloaded functions.

If Γ is a typing context, we say that an environment η *satisfies* Γ if $\eta(x) \in [\![\sigma]\!]\eta$, for every $x : \sigma \in \Gamma$. The notation $\eta \models \Gamma$ abbreviates "$\eta(x) \in [\![\sigma]\!]\eta$, for every $x : \sigma \in \Gamma$".

The meaning of a term e in an environment η is defined by induction on typing derivations $\Gamma \vdash e : \sigma$, for typing contexts Γ such that $\eta \models \Gamma$, as follows:

$$[\![\Gamma, x : \sigma \vdash x : \sigma]\!]\eta = \eta(x)$$

$$[\![\Gamma \vdash e\ e' : \tau]\!]\eta = \mathbf{App}^{\tau',\tau}\ ([\![\Gamma \vdash e : \tau' \to \tau]\!]\eta)\ ([\![\Gamma \vdash e' : \tau']\!]\eta)$$

$$[\![\Gamma \vdash \lambda u.\, e : \tau \to \tau']\!]\eta =$$

the unique $f \in A \to B$, where $A = [\![\tau]\!]\eta$ and $B = [\![\tau']\!]\eta$, such that

for all $a \in A$, $\mathbf{App}^{\tau,\tau'}\, f\ a = [\![\Gamma, u : \tau \vdash e : \tau']\!]\eta[u \mapsto a]$

$$[\![\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : \tau]\!]\eta =$$

if $x \in U$ then $[\![\Gamma, x : \sigma \vdash e' : \tau]\!]\eta[x \mapsto d]$

else $[\![\Gamma, x : \sigma \vdash e' : \tau]\!]\eta'$

where $d = [\![\Gamma \vdash e : \sigma]\!]\eta$

$$\eta' = \eta[x \mapsto \mathit{extend}(\eta(x), d)]$$

$$\mathit{extend}(f, g) = \lambda x.\ \text{if } x \in \mathit{dom}(g) \text{ then } g(x) \text{ else } f(x)$$

6 Conclusion

In extensions of the Damas-Milner type system, class types, as presented in this paper, provide a simplified treament of overloading. In system CT, the overloading policy is controlled by means of a predicate used in a single inference rule. Types can be inferred, without the need for any type annotations, and there is an algorithm for computing most general typings. The use of type annotations can simplify inferred types, though.

System CT has less restrictions than existing similar systems that extend ML-like polymorphic type systems with context-independent overloading. For example, in System CT types of overloaded functions can be such that they have the same argument type constructor.

More experience is needed on the use of overloading and class types together in a system with polymorphism and type inference. The main motivation for our study came out from the idea of defining a form of higher-order types, that can be used as types of (lower-order) abstract and concrete types. Lower-order types would be instances of higher-order types. We intend to explore the use of class types together with higher-order types, following the idea that higher-order types are types parameterised on other, possibly constrained types. We think this idea can enhance a functorial view of modules as parameterised types.

Further explorations of this system involve incorporating the concept of class types in a functional programming language, like Haskell or SML, studying the implications of this concept with regards to the subtyping relation and program development, and studying its relation to intersection types [12].

References

[1] Damas, Luis and Robin Milner. Principal type schemes for functional programs. *Proc. 9th ACM Symp. on Principles of Programming Languages*, pages 207–212, 1982.

[2] Hall, C., K. Hammond, S.P. Jones and P. Wadler. Type Classes in Haskell. *Proc. 5th European Symposium on Programming*, pages 241–256, 1994. Springer LNCS 788.

[3] Jones, Mark, *Qualified Types*. Cambridge University Press, 1994.

[4] Jones, Simon Peyton et al., Report on the Programming Language Haskell. *ACM SigPlan Notices*, 27(5), 1992.

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

[6] Milner, Robin, Mads Tofte and Robert Harper. *The Definition of Standard ML*. MIT Press, 1989.

[7] Mitchell, J.C., *Foundations for programming languages*. MIT Press, 1996.

[8] Odersky, M., P. WadlerM. and M. Wehr. A Second Look at Overloading. *Conference Record of Functional Programming and Computer Architecture*, 1995.

[9] Paulson, L.C., *ML for the Working Programmer*. Cambridge University Press, 1996. 2nd edition.

[10] Simon Thompson. *Haskell: The Craft of Functional Programming.* Addison-Wesley, 1996.

[11] Turner, D. A., A non-strict functional language with polymorphic types. In *Proceedings of the 2nd International Conference on Functional Programming and Computer Architecture.* IEEE Computer Society Press, 1985.

[12] van Bakel, S., Essential Intersection Type Assignment. In R.K. Shyamasunda, editor, *Proc. 13th Conf. Foundations of Software Technology and Theoretical Computer Science*, LNCS **761** (1993), pp. 13–23.

[13] Wadler, Philip, How to make *ad-hoc* polymorphism less *ad hoc*. *Conf. Record of the 16th ACM Symposium on Principles of Programming Languages*, 1989.