Parametric Overloading in Polymorphic Programming Languages

Stefan Kaes

Fachbereich Informatik

Technische Hochschule Darmstadt

Magdalenenstr. 11c, D-6100 Darmstadt

BITNET: xlp2fgpl@ddathd21

Abstract

The introduction of unrestricted overloading in languagues with type systems based on implicit parametric polymorphism generally destroys the principal type property: namely that the type of every expression can uniformly be represented by a single type expression over some set of type variables. As a consequence, type inference in the presence of unrestricted overloading can become a NP-complete problem. In this paper we define the concept of parametric overloading as a restricted form of overloading which is easily combined with parametric polymorphism. Parametric overloading preserves the principal type property, thereby allowing the design of efficient type inference algorithms. We present sound type deduction systems, both for predefined and programmer defined overloading. Finally we state that parametric overloading can be resolved either statically, at compile time, or dynamically, during program execution.

1 Introduction

Over the last decade, a considerable number of (functional) programming languages with type disciplines based on the concept of parametric polymorphism [Milner78] have been developed. Among the better known are ML [Milner84], HOPE [BMQS80] and Miranda [Turner85]. The success of parametric polymorphism is largely due to the following facts.

Security: Programs are forced to be statically type correct, enabling the detection of a large number of programming errors at compile time.

Flexibility: Parametric polymorphism provides the programmer with a type system which allows the reuse of functions for arguments of various types, provided the meaning of the function does not depend on a particular type.

Efficiency: Since all type checking is done at compile time, expensive runtime type checks can be eliminated, thus increasing the efficiency of functional language implementations.

At a closer look, however, one can observe that the type systems employed in these languages are

not as secure and flexible as they should be. For example, let us consider some predefined operators in the context of a type system containing integers, reals, booleans, lists of homogenous element type and functions.

A typical example for a parametrically polymorphic function is the function *len*, which determines the length of lists of arbitrary element type *t*. In a parametrically polymorphic type system the type of *len* could be expressed using the type expression $\forall t.\ list(t) \rightarrow int$, where *t* stands for an arbitrary type.

In regard to flexibility, we would certainly like to have an operator +, to denote both integer- and real addition at the same time. Thus, we require + to be *overloaded*. In a parametrically polymorphic type system, this can only be described by assigning a set of type expressions, (equivalent to the conjunctive types of [Coppo80]) as the type of +, namely: $\{\ int \times int \rightarrow int\ ,\ real \times real \rightarrow real\ \}$.

This presents no conceptual problems for type inference, since it is well known how to deal with this kind of overloading (see [BaSne86] for a possible approach). However, in the presence of arbitrary sets of types and undeclared identifiers, the problem becomes computationally hard to solve, in fact, it can be shown to be NP-complete [ASU86]. It appears as if this problem has led the designers of HOPE, Miranda and ML(at least in the original version) to the decision of collapsing types integer and real into a single data type of numbers. Nonetheless, we feel that the distinction of integer and real numbers is essential and should therefore be reflected in the type system of any programming laguage.

Note that there is also a semantical difference between overloading and polymorphism: In the case of the overloaded + operator, we would expect different code to be executed for integer and real addition respectively, whereas we expect the code of *len* to be usable for lists of any type. This distinction has led to the name "adhoc polymorphism" [Str67]. At a second glance it is somewhat superficial: we could easily imagine a built-in addition instruction, which tests the type of the operands and then performs either integer or real addition.

As a striking example for the insecurity of pure parametric polymorphism, let us assume that = , the equality operator, is applicable to two values of the same type, provided there is no comparison between functional values involved.

The reason for this restriction is rather obvious: since equality of functions is in general undecidable, we would like the type system to prohibit any attempt to do so. Using type expressions over some set of ordinary type variables, one can only assign $\forall t.\ t \times t \rightarrow bool$ as a polymorphic type to =. This type expression can be interpreted as: for any type t, = maps two values of type t to a boolean value. Since any type is admissible, this implies that even two functions of identical type, say $int \rightarrow int$, are comparable.

Now this example has a flavor not found in the first one: if we were to represent the overloaded type of = by a set of type expressions, we would be forced to use an infinite number of types:

$$\{\ int \times int \rightarrow bool, \ldots,\ list(int) \times list(int) \rightarrow bool, \ldots,\ list(list(int)) \times list(list(int)) \rightarrow bool, \ldots\ \}$$

We conclude therefore that it is necessary to introduce a finite representation for such sets, especially since these sets arise frequently, some examples being the sets of types of =, ≤, etc. . Such a representation is possible, if we restrict ourselves to *parametric overloading*. The difference between the various kinds of polymorphism can roughly be summarized by the following comparison:

Parametric polymorphism: One semantic object can have different types at each usage, all being

instances of a single type expression over variables.

Overloading ("adhoc polymorphism"): A single name can be used to denote different semantic objects, the types of these objects being completely unrelated.

Parametric overloading: A single name can be used to denote several objects, the types of these objects being instances of a single type expression over some extended set of type variables.

The remainder of this paper is devoted to the development of a theory of parametric overloading. It is organized in the following way: First, we recapitulate the theoretical foundations of the notion of parametric polymorphism in the context of a simple expression language. We present a denotational semantics for this language, define syntax and semantics of types, give a deduction system for inferring well typed programs, and state that our type system is sound w.r.t. the semantics of expressions and types. Second, we give a formal definition of parametric overloading, based on the concept of overloading assumptions and sorted type variables, restricting ourselves to the case of predefined overloaded function symbols. We then proceed to show that a unification algorithm for the modified set of type expressions exists, and that, by simply replacing robinsons algorithm used in Milner's type inference algorithm W with this new algorithm, we obtain an algorithm to compute principal well typings in presence of parametrically overloaded functions. In Chapter 4 we extend our base language to include user definable overloading, and give a deduction system for well typings. Finally we discuss two alternative semantics for the extended language: the first one is given by mapping overloaded expressions back to expressions of the original language, resolving overloading statically, the second one is given by changing the semantic equations to resolve overloading dynamically, during program execution.

2 Parametric Polymorphism

2.1 Syntax and Semantics of Expressions

Assuming that x ranges over a countable set of variables, the expressions of our example language **Expr** are generated by the grammar

$$M ::= x \mid \lambda x.M \mid M_1\ M_2 \mid \textbf{let}\ x = M_1\ \textbf{in}\ M_2$$

In order to assign meaning to expressions we postulate the existence of a domain of denotable values **V** as a solution to the recursive domain equation

$$\mathbf{V} = \mathbf{W} \oplus \left(\bigoplus S_c(\mathbf{V}^{a(c)}) \right)$$

where $\oplus$ denotes disjoint union and

(1) $\mathbf{W} = \{ \cdot \}$ is the domain of the value $\cdot$, used to model runtime type errors, where $\cdot$, as a member of **V** will be denoted by *wrong.*

(2) C is a finite set of type constructors with arity a(c); e.g. {int, real, $\times$, $\rightarrow$, ... }

(3) $S_c(D_1,\ldots,D_{a(c)})$ is a domain corresponding to type constructor $c \in C$; e.g.:

S_{int}	- the flat cpo of integers
S_{real}	- likewise for real numbers
$S_\times(A,B)$	- cartesian product space
$S_\rightarrow(A,B)$	- space of continous function from A to B

The meaning of any expression M is given in terms of a function $\mathcal{E}[\![M]\!]$, mapping environments $\eta \in \mathbf{Env} = \mathbf{Id} \rightarrow \mathbf{V}$ to denotable values:

$\mathcal{E}$: **Expr** → **Env** → **V**
$\mathcal{E}[\![x]\!]\eta = \eta(x)$
$\mathcal{E}[\![\lambda x.M]\!]\eta = (\lambda v.\ \mathcal{E}[\![M]\!]\eta\ \{v/x\})$
$\mathcal{E}[\![M_1\ M_2]\!]\eta$ = *if* f ∈ F *then* (f | F) v *else wrong* , where f = $\mathcal{E}[\![M_1]\!]\eta$, v = $\mathcal{E}[\![M_2]\!]\eta$
$\mathcal{E}[\![$**let** x = M_1 **in** $M_2]\!]\eta = \mathcal{E}[\![M_1]\!]\ \eta\{\ \mathcal{E}[\![M_2]\!]\eta/x\ \}$

where F denotes the function space $S_{\to}(V,V)$ and $\eta\{\ v/x\ \} \equiv \lambda y.$ *if* x=y *then* v *else* $\eta(y)$.

Note that the only possible type error apparent from the equations above, is the application of non-functions to argument values. However, we may assume that application of functions in η to arguments outside their domain can cause type errors as well.

2.2 Syntax and Semantics of Type Expressions

Let **Tvars** = {α,β,...} be a countable set of *type variables* and c range over C. The syntax of *types* τ and *type-schemes* σ is defined by

$$\tau ::= \alpha \mid c(\tau_1,...,\tau_n) \qquad \text{and} \qquad \sigma ::= \tau \mid \forall\alpha.\sigma$$

The variables $\alpha_1,..,\alpha_n$ in type-scheme $\forall\alpha_1,..,\alpha_n.\tau$ (an abbreviation of $\forall\alpha_1.\ ...\ \forall\alpha_n.\ \tau$) are called *generic*, whereas any type variable free in τ is called *specific*. *Monotypes* t are types not containing type variables. Substitutions S ∈ **Subst** = **Tvars** $\overrightarrow{m}$ **Types** are finite mappings of type variables to types. If σ is a type-scheme, S = $[\tau_i/\alpha_i]$ ∈ **Subst**, then S(σ) is the type-scheme obtained by instantiating each free occurence of α_i in σ to τ_i, where bound variables in σ may be renamed to avoid name clashes (cf. α-conversion). A type scheme σ' = $\forall\beta_1,...,\beta_n.\tau'$ is called *generic instance* of σ (written σ' < σ), if σ = $\forall\alpha_1,...,\alpha_n.\tau$ and τ' = $[\tau_1/\alpha_1,...,\tau_n/\alpha_n]\tau$ and no β_i is free in τ.

Following [MQPS84] types can be interpreted as elements in the lattice of *weak ideals* **I(V)**. Ideals I⊆V do not contain *wrong* , are non-empty, downward closed and limit closed. Moreover, **I(V)** is closed under union and intersection. For any c ∈ C of arity n, let $\mathcal{i}$(c)($\mathbf{I}_1,...,\mathbf{I}_n$) denote the image of $S_c(\mathbf{I}_1,...,\mathbf{I}_n)$ in **V**. For example, let $\mathcal{i}$(int) = S_{int} , $\mathcal{i}$(real) = S_{real} , $\mathcal{i}$(×)(**I**,**J**)={ ⟨a,b⟩ | a ∈ A, b ∈ B } and $\mathcal{i}$(→)(**I**,**J**) = { f ∈ **V**→**V** | x ∈ **I** ➡ f(x) ∈ **J** }.

Let φ ∈ **Tenv** = **Tvars** $\overrightarrow{m}$ **I(V)**, then function $\mathcal{T}$ below maps type schemes σ to ideals under an assignment φ of ideals to type variables free in σ:

$\mathcal{T}[\![\alpha]\!]\varphi = \varphi(\alpha)$
$\mathcal{T}[\![c(\tau_1,...,\tau_n)]\!]\varphi = \mathcal{i}(c)(\mathcal{T}[\![\tau_1]\!]\varphi,...,\mathcal{T}[\![\tau_n]\!]\varphi)$
$\mathcal{T}[\![\forall\alpha.\sigma]\!]\varphi = \bigcap_{t\in\mathbf{Types}} \mathcal{T}[\![\sigma]\!]\ \varphi\{\ \mathcal{T}[\![t]\!]\varphi/\alpha\ \}$

2.3 Well-typed Expressions

A *typing* is a statement of the form A ⊢ M : τ, where A, a finite set of pairs x:σ, is called *type assumption*, assigning types to variables occuring free in M. If A is a type assumption, then A. x:σ denotes A ∪ {x:σ} and A_x denotes A, except that x does not occurr in A_x. $\overline{A\tau}$, the closure of τ with respect to A is defined as $\overline{A\tau} =_{def} \forall\alpha_1,...,\alpha_n.\tau$, where $\{\alpha_1,...,\alpha_n\}$ is the largest set of type variables which occur free in τ, but not in A.

Typing A ⊢ M : τ is said to be a *well-typing* if it can be derived using the following axioms and

inference rules:

$$[\text{VAR}]\quad A_x.\ x{:}\,\sigma \vdash x : \tau \qquad \text{if } \tau < \sigma$$

$$[\text{ABS}]\quad \frac{A_x.\ x : \tau_a \vdash M : \tau_r}{A \vdash \lambda x.M : \tau_a \to \tau_r}$$

$$[\text{APP}]\quad \frac{A \vdash M_1 : \tau_a \to \tau_r\ ,\ A \vdash M_2 : \tau_a}{A \vdash M_1\ M_2 : \tau_r}$$

$$[\text{LET}]\quad \frac{A \vdash M_1 : \tau_1\ ,\ A_x.\ x{:}\ \overline{A\tau_1} \vdash M_2 : \tau_2}{A \vdash \textbf{let } x = M_1 \textbf{ in } M_2 : \tau_2}$$

Note that this system differs from the one given in [DaMi82] in two aspects: instantiation of type schemes ($<$) is restricted to variable usage and generalisation of types ($\overline{A\tau}$) is restricted to let-introduced identifiers. The system was first used in [CDDK86], where it was shown to be equivalent to the system of [DaMi82] in the following sense: Every well-typing derivable in the system above is also derivable in the system of [DaMi82]. Conversely, if $A \vdash M : \sigma$ is derivable in the system of [DaMi82], then there exists a well-typing $A \vdash M : \tau$, derivable in the above system, such that $\sigma < \overline{A\tau}$.

The following two theorems state the key properties of well-typings, namely, that well-typed expressions do not produce runtime type errors and that type inference can be used to compute principal types:

Theorem1 (Milner): *soundness of type inference*

If $A \vdash M : \tau$ is a well-typing and $x{:}\sigma \in A \Rightarrow \eta(x) \in \mathcal{T}[\![\sigma]\!]\varphi$ then $\mathcal{E}[\![M]\!]\eta \in \mathcal{T}[\![\overline{A\tau}]\!]\varphi$.

Theorem2 (Hindley-Damas-Milner): *principal well-typings*

Let A be a type assumption, A' an instance of A and FV(M) be the variables free in M. If $A' \vdash M : \tau'$ is a well-typing, then there exists a principal well-typing $A \vdash M : \tau$, such that there exists a substitution S satisfying $A'|_{FV(M)} = S(A|_{FV(M)})$ and $\overline{A'\tau'} < \overline{SAS\tau}$. Moreover, there exists an algorithm (Algorithm W) to compute $A \vdash M : \tau$.

3 Parametric Overloading

3.1 Overloading Schemes and Overloading Assumptions

In this chapter we will to introduce the concept of parametric overloading, which can be characterised by a set of restrictions on ordinary overloading:

- overloading is restricted to function symbols (identifiers)
- the result type of a function application f(x,...,z) is uniquely determined by the outermost type constructor of the argument types

Moreover, in order to make type inference feasible, we require that the set of types of an overloaded function can always be represented by a pair consisting of a designated type expression called *overloading scheme* and a set of instantiation rules both of which together are called *overloading assumption.*

Definition: *overloading scheme*

Let \$ be a special symbol not found in $C \cup Tvars$, then ω is called an overloading scheme **iff** $\omega = \omega_0 \times \ldots \times \omega_{n-1} \to \omega_n$ where for all $i \in 0..n$, ω_i is either a type τ, or the special symbol \$, and $\omega_{n-1} = \$ \Rightarrow \exists\ i \in 0..n-1$ such that $\omega_i = \$$.

In any overloading scheme $ designates the argument positions which may be overloaded. Examples of overloading schemes are:

$ → int	a discrete measure
$ → real	a continuous measure
$ → $	succ, pred
$ × $ → bool	=, ≠, ≤, ≥, <, >
$ × $ → $	+, *, −, ∧, ∨

Let ω be an overloading scheme and τ some type expression, then:

τ overloads ω with τ' **iff** $\tau = [\tau'/\$]\omega$ and τ' is of the form $c(\alpha_1,...,\alpha_n)$ for some $c \in C$.

In this case we also say that τ' is an overloading for ω. The restriction on the form of τ' is due to the fact that we want to resolve overloading by looking at outermost type constructors only.

Definition: *overloading assumption*

An overloading assumption O is a finite set of pairs $x:\langle\omega,s\rangle$, where each x occurs only once in O , ω is an overloading scheme and s is a set of overloadings for ω, such that no type constructor occurs twice in s.

Using this definition we could now define the set of valid overloadings of $x:\langle\omega,s\rangle$ to be the set of types t such that $t = [t'/\$]\omega$ where t' is an instance of some type constructor $c \in s$.

The overloading assumption { +: ⟨$ × $ → $, { int, real } } would then specify the type set { int × int → int, real × real → real }. However, if we try to define the possible overloadings of the predefined equality operator using the assumption

{ =: ⟨$ × $ → $, { int, real, list(α) }⟩},

we still get too large a set: although we have effectively excluded functions from appearing as arguments of =, we still allow lists of functions to be compared. In order to remedy this situation, we refine our notion of type variables: type variables come equipped with a set of operator names X, where intuitively, α_X stands for the set of all types that can appear at an overloaded argument position of every operator $x \in X$. The overloading assumption

{ =: ⟨$ × $ → $, { int, real, list($\alpha_{(=)}$) } ⟩},

would then restrict the type of arguments for = to all types constructed from the type constructors int, real and list.

Definition: *$x \not\vdash \tau$, $X \not\vdash \tau$, valid overloadings*

Let O be an overloading assumption. If α is a type variable marked with a set of operator names, then let ops(α) denote that set. Type τ can appear at an overloaded argument position of operator x (written $x \not\vdash \tau$) **iff** either $\tau \in Tvars \wedge x \in ops(\tau)$ or $\tau = c(\tau_1,...,\tau_n) \wedge O = O_x.\ x:\langle\omega,\{...\ c(\alpha_1,...,\alpha_n)\ ...\}\rangle \wedge \forall\ i=1..n:\ \forall y \in ops(\alpha_i):\ y \not\vdash \tau_i$.

If X is a set of operators, then $X \not\vdash \tau \equiv \forall x \in X.\ x \not\vdash \tau$. The set of valid overloadings of x is given by $\{ [t/\$]\omega \mid x \not\vdash t \}$.

Example: Suppose we want to impose the following restrictions on the set of valid overloadings of the operators +, = and ≤ : equality is defined for integers, reals, lists and sets, provided set and list elements can be compared. + is overloaded with integer and real addition, and set union, whereas ≤ is used to denote the arithmetic ≤-relation, the sublist- and subset relation. These restrictions are correctly specified by the overloading assumption

$+ : \langle \$ \times \$ \to \$, \{ \text{int, real, set}(\alpha_{\{=\}}) \} \rangle$

$= : \langle \$ \times \$ \to \text{bool}, \{ \text{int, real, list}(\alpha_{\{=\}}), \text{set}(\alpha_{\{=\}}) \} \rangle$

$\leq : \langle \$ \times \$ \to \text{bool}, \{ \text{int, real, list}(\alpha_{\{=\}}), \text{set}(\alpha_{\{=\}}) \} \rangle$

3.2 Well-typed Expressions

Let A be a type assumption and O be an overloading assumption. We say that A agrees with O **iff** whenever $x{:}\langle\omega,s\rangle \in O$ and $x{:}\sigma \in A$ then $\sigma = [\alpha_{(x)}/\$]\omega$. Let $<_O$ be the generic instance relation respecting $\not\in$ and let us assume that predefined operators cannot be redeclared. The following deduction system can be used to infer valid typings in the presence of a fixed overloading assumption:

[VAR] $A_x.\ x{:}\sigma \not\in^{O} x : \tau$ if $\tau <_O \sigma$

[ABS] $\dfrac{A_x.\ x : \tau_a \not\in^{O} M : \tau_r}{A \not\in^{O} \lambda x.M : \tau_a \to \tau_r}$ if not $x \in O$

[APP] $\dfrac{A \not\in^{O} M_1 : \tau_a \to \tau_r \ ,\ A \not\in^{O} M_2 : \tau_a}{A \not\in^{O} M_1\ M_2 : \tau_r}$

[LET] $\dfrac{A \not\in^{O} M_1 : \tau_1 \ ,\ A_x.\ x{:}\overline{A\tau_1} \not\in^{O} M_2 : \tau_2}{A \not\in^{O} \textbf{let } x = M_1 \textbf{ in } M_2 : \tau_2}$ if not $x \in O$

Note that the main difference between this system and the one given for nonoverloaded expressions is the restriction that generic instantiation has to respect $\not\in$.

Example: Using the system above one can deduce, that given type assumption

$A = \{ +{:}\ \forall\alpha_{\{+\}}.\ \alpha_{\{+\}} \times \alpha_{\{+\}} \to \alpha_{\{+\}},\ *{:}\ \forall\alpha_{\{*\}}.\ \alpha_{\{*\}} \times \alpha_{\{*\}} \to \alpha_{\{*\}} \},$

and overloading assumption

$O = \{ +{:}\ \langle \$ \times \$ \to \$, \{ \text{int, real} \} \rangle,\ *{:}\ \langle \$ \times \$ \to \$, \{ \text{int, real} \} \rangle \}$

the typing

$A \not\in^{O} \textbf{let } f = \lambda x.\ x + x * x \textbf{ in } (f\ 5) : \text{int}$

is a well typing. Note that the declaration of f causes the type-scheme assigned to f to be

$\forall\alpha.\alpha_{\{+,*\}} \times \alpha_{\{+,*\}} \to \alpha_{\{+,*\}}$

which says that f is a overloaded function applicable to values to which both + and * are applicable simultaneously. The type of f in the application (f 5) is then instantiated to int x int → int, since int is a valid instantiation for $\alpha_{\{+,*\}}$.

Proving a soundness theorem for the modified deduction system requires an adjustment to the semantic function $\mathcal{T}$, taking into account that variables are sorted w.r.t. operator names. For a fixed overloading assumption O and $\varphi \in \textbf{Tenv}$, such that $\alpha_s \in \text{dom } \varphi \Rightarrow \exists t.\, s \not\in t \wedge \varphi(\alpha_s) = \mathcal{T}_O[\![t]\!]\emptyset$, we define $\mathcal{T}_O$ as:

$\mathcal{T}_O[\![\alpha_s]\!]\varphi = \varphi(\alpha_s)$

$\mathcal{T}_O[\![c(\tau_1,\ldots,\tau_n)]\!]\varphi = \iota(c)(\mathcal{T}_O[\![\tau_1]\!]\varphi,\ldots,\mathcal{T}_O[\![\tau_n]\!]\varphi)$

$\mathcal{T}_O[\![\forall\alpha_s.\sigma]\!]\varphi = \bigcap_{\substack{t \in \textbf{Types} \\ s \not\in t}} \mathcal{T}_O[\![\sigma]\!]\ \varphi\{\ \mathcal{T}_O[\![t]\!]\varphi/\alpha\ \}.$

In order for $\mathcal{T}_O$ to be well defined we have to request that for every set of operator names s, the set $\{ t \mid s \not\in t \}$ is always nonempty. This can be achieved simply by adding a nonsense type contructor ? of zero arity to C, such that $x \not\in ?$ holds for every operator x. Additionally, the domain of denotable

values is extended by the single valued domain $S_? = \{ ? \}$. Note that the value ? can never be denoted by expressions. This construction is only necessary in order to simplify the statement and proof of soundness theorems to follow.

Theorem: *soundness of type inference in the presence of predefined overloaded operators:*

If $A \vdash^{O} M : \tau$ is a well-typing, A agrees with O and $x{:}\sigma \in A \Rightarrow \eta(x) \in \mathcal{T}_O[\![\sigma]\!]\varphi$ then $\mathcal{E}[\![M]\!]\eta \in \mathcal{T}_O[\![\overline{A\tau}]\!]\varphi$.

(Note: the proof of this and all other theorems in this paper can be found in [Kaes87].)

3.3 Type Inference in the Presence of a Fixed Overloading Assumption

In order to develop a type inference algorithm which computes principal typings for our modified deduction system, all we need to do is, find an unification algorithm for type expressions over variables marked with operator names, replace it for the Robinson algorithm in Milner's algorithm W and we are done ! We will therefore concentrate on the presentation of the new unification algorithm.

For sake of simplicity, for the rest of the paragraph we assume that O is a fixed over loading assumption. Then, let the functions $m: C \to 2^{Id}$ and $d: C \times Nat \times 2^{Id} \to 2^{Id}$ be defined in the following way:

$$m(c) =_{def} \{ x \mid x: \langle \omega, \{ \ldots c(\ldots) \ldots \} \rangle \in O \}$$

$$d_c(i, X) =_{def} \bigcup_{x \in X} \{ ops(\alpha_i) \mid x: \langle \{ \ldots c(\alpha_1, \ldots, \alpha_n) \ldots \} \rangle \in O \}$$

m(c) maps type names to the set of operators, where such a type name may possibly occur at an overloaded argument position. $d_c(i,X)$ maps sets of type names to the set Y_i, such that if $Y_i \Vdash \tau_i$ holds for i=1..n and $X \subseteq m(c)$ then $X \Vdash c(\tau_1, \ldots, \tau_n)$ holds as well.

Let S be a substitution. We say that S respects variable sorts if $\alpha \in dom\ S \Rightarrow ops(\alpha) \Vdash S(\alpha)$. Let τ_1 and τ_2 be types. τ_2 is a valid instance of τ_1 ($\tau_1 \geq \tau_2$), if there exists a substitution S, respecting variable sorts, such that $\tau_2 = S\tau_1$. If S_1 and S_2 are substitutions, S_1 is more general than S_2 ($S_1 \geq S_2$), if $dom\ S_1 \subseteq dom\ S_2$ and $\alpha \in dom\ S_1 \Rightarrow S_1(\alpha) \geq S_2(\alpha)$.

Lemma: *most general substitutions*

Let τ be a type and X be a set of operator names. Then there exists either no substitution satisfying $X \Vdash \tau$ or a most general one. Moreover, the algorithm cs given below can be used to compute such a substition.

$cs(X, \alpha) = [\beta/\alpha]$	β a new type variable, $ops(\beta) = s \cup ops(\alpha)$
$cs(X, c(\tau_1, \ldots, \tau_n)) = S_n$	if $X \subseteq m(c) \wedge \exists\ S_0 \ldots S_n$ such that $\forall\ i=1..n: S_i = cs(d_c(i,X), S_{i-1}(\tau_i)) \bullet S_{i-1}$

cs fails in all other cases.

Proof: By computational induction one can show that (i) if cs succeeds, it will return a substitution S which respects variable sorts and (ii) for any other substitution R which satisfies (i) one can find a substition S' such that $R = S' \bullet S$.

As an immediate consequence of the existence of most general substitutions respecting variable sorts we get the following

Theorem: *most general unifiers*

Given types τ_1 and τ_2 there exists either no unifier S, such that $S(\tau_1) = S(\tau_2)$, or a

most general one. Moerover, if a mgu of τ_1 and τ_2 exists, it can be computed by the unification algorithm $\mathcal{U}$ below:

$\mathcal{U}(\tau, \tau) = [\,]$

$\mathcal{U}(\tau, \alpha) = \mathcal{U}(\alpha, \tau)$ if not $\tau \in Tvars$

$\mathcal{U}(\alpha, \tau) = [S(\tau)/\alpha] \bullet S$ if $S = cs(ops(\alpha), \tau)$ exists and not $\alpha \in vars(\tau)$

$\mathcal{U}(c(\tau_1,...,\tau_n), c(\tau_1',...,\tau_n')) = S_n$ if $\exists\ S_0,...,S_n.\ S_0 = [\,] \wedge \forall\ i=1..n:$
$S_i = \mathcal{U}(S_{i-1}(\tau_i), S_{i-1}(\tau_i')) \bullet S_{i-1}$

$\mathcal{U}(\tau, \tau')$ fails in all other cases.

Proof: Computational induction using the lemma above.

Observe that the algorithms cs and $\mathcal{U}$ are quite independend of the actual semantics behind the functions m and d and are therefore more general than appears at first sight. For example, given appropriate definitions of d and m, one can obtain an unfication algorithm for *order sorted algebras* from $\mathcal{U}$ and cs (cf. the algorithm in [BaSne86]). Details of the construction may be found in [Kaes87].

4 User Definable Parametric Overloading

4.1 Syntax and Type Deduction

In this paragraph we investigate the effect of extending our base language to enable programmers to define their own overloaded function symbols. We add two clauses:

$M ::= x \mid \lambda x.M \mid M_1\ M_2 \mid$ **let** $x = M_1$ **in** M_2
$\mid$ **letop** x: ω **in** M
$\mid M_1 : \sigma$ **extends** x **in** M_2

The letop-clause declares x as an identifier with overloading scheme ω, overloadable in M, whereas the extend-clause overloads x with the meaning of M_1 in M_2, provided σ is a valid overloading of overloadable operator x.

As an example, the following program defines the usual overloading of multiplication under the assumption that pair: $\forall\alpha,\beta.\alpha\to\beta\to(\alpha,\beta)$ contructs pairs, fst:$\forall\alpha,\beta.(\alpha,\beta)\to\alpha$ and snd:$\forall\alpha,\beta.(\alpha,\beta)\to\beta$ select the first resp. second components of pairs:

letop *: $ × $ → $ **in**
intmult: int × int → int **extends** * **in**
realmult: real × real → real **extends** * **in**
let addsquares = λp. fst(p)*fst(p)+snd(p)*snd(p) **in**
addsquares (pair 3 5)

It is not very surprising that we can adapt our type deduction system to the new situation, by moving the overloading assumption, under which we infer valid types, to the assumption part of our typings. Therefore $A \overset{O}{\vdash} M : \tau$ now becomes $\langle O, A\rangle \vdash M : \tau$, which can be read as: under overloading assumption O and type assumption A for identifiers free in M we can derive that τ is a valid type for M. This leads to the following deduction system:

[VAR] $\langle O, A_x.\ x: \sigma \rangle \vdash x : \tau$ if $\tau <_O \sigma$

$$[\text{ABS}]\quad \frac{\langle O_x, A_x.\ x{:}\ \tau_a \rangle \vdash M : \tau_r}{\langle O, A \rangle \vdash \lambda x.M : \tau_a \to \tau_r}$$

$$[\text{APP}]\quad \frac{\langle O, A \rangle \vdash M_1 : \tau_a \to \tau_r\ ,\ \langle O, A \rangle \vdash M_2 : \tau_a}{\langle O, A \rangle \vdash M_1\ M_2 : \tau_r}$$

$$[\text{LET}]\quad \frac{\langle O, A \rangle \vdash M_1 : \tau_1\ ,\ \langle O_x, A_x.\ x{:}\ \overline{A\tau_1} \rangle \vdash M_2 : \tau_2}{\langle O, A \rangle \vdash \textbf{let}\ x = M_1\ \textbf{in}\ M_2 : \tau_2}$$

$$[\text{OP}]\quad \frac{\langle O_x.\ x{:}\ \langle \omega, \{?\} \rangle, A_x.\ x{:}\ \forall \alpha_{\{x\}}.[\alpha_{\{x\}}/\$]\omega \rangle \vdash M : \tau}{\langle O, A \rangle \vdash \textbf{letop}\ x{:}\ \omega\ \textbf{in}\ M : \tau} \qquad \text{if } \beta_s \in \text{vars}(\tau) \Rightarrow x \notin s$$

$$[\text{OV}]\quad \frac{\langle O_x.\ x{:}\langle \omega, d \rangle, A \rangle \vdash M_1 : \sigma \quad \langle O_x.\ x{:}\langle \omega, d \cup \{\tau\} \rangle, A \rangle \vdash M_2 : \tau'}{\langle O_x.\ x{:}\langle \omega, d \rangle, A \rangle \vdash M_1{:}\ \sigma\ \textbf{extends}\ x\ \textbf{in}\ M_2 : \tau'} \qquad \begin{array}{l}\text{if } \sigma \text{ overloads } \omega \text{ with } \tau,\\ \tau = c(\alpha_1,\ldots,\alpha_n) \wedge \text{not } c \in d\end{array}$$

The condition $\beta_s \in \text{vars}(\tau) \Rightarrow x \notin s$ in the OP-rule ensures, that no overloading can be exported out of its scope, whereas the condition in OV guarantees, that no type constructor can appear twice in any overloading set.

Theorem: *principal well-typings in the presence of user definable overloading*

If $\langle O,A' \rangle \vdash \tau'$ is a well typing in the system above and A' an instance of A, then there exists a principal well-typing $\langle O,A \rangle \vdash M : \tau$ such that there exists a substitution S satisfying $A'|_{FV(M)} = S(A|_{FV(M)})$ and $\overline{A'\tau'} <_O \overline{SAS\tau}$. Moreover, there exists an algorithm (Algorithm Z) to compute $\langle O,A \rangle \vdash M : \tau$.

Algorithm Z is a rather straightforward extension of algorithm W, obtained by adding the overloading assumption as an extra parameter and including some functions to manipulate it in the appropriate way. For details the reader is again referred to [Kaes87].

Although type inference remains relatively simple, the semantics of expressions gets rather complicated. We can try two approaches: Static overloading resolution, through removal of every occurrence of letop- or extend clauses from our expessions, or a more direct semantics, extending semantic domains and equations, thus enabling dynamic overloading resolution.

4.2 Static Overloading Resolution

Static overloading resolution aims at execution efficiency by avoiding any type checking necessary for overloading resolution at runtime. It does not come for free though, complicating compilation through the need of (possibly costly) compile time program transformations. In this paragraph we present a overloading resolution function $\mathcal{R}$, mapping well typed overloaded expressions back to expressions of our original language.

Let $\langle O,A \rangle \vdash M : \tau$ be a well typing and M^τ be the well typed expression M where all its sub-expressions are annotated with their type. Let $\rho \in \textbf{Renv} = \textbf{TypedId} \to \textbf{Expr}$ be an overloading resolution environment and let $\rho \backslash x$ denote ρ with all occurences of x removed, then function $\mathcal{R}$: **Expr → Renv → Expr** removes any user defined overloading from M:

$$\mathcal{R}[\![x^\tau]\!]\ \rho = \textit{if}\ \exists\ x^\sigma \in \text{dom}\ \rho.\ \sigma = \forall \alpha_i.\tau'\ ,\ \tau = S(\tau'),\ S \text{ overloading resolving}$$
$$\textit{then}\ \mathcal{R}[\![S(\rho(x^\sigma))]\!]\ \rho\ \textit{else}\ x^\tau$$

$\mathcal{R}[\![(M_1\ M_2)^\tau]\!]\ \rho = (\mathcal{R}[\![M_1]\!]\rho\ \ \mathcal{R}[\![M_2]\!]\rho\ )^\tau$

$\mathcal{R}[\![(\lambda x.M)^\tau]\!]\ \rho = (\ \lambda x.\mathcal{R}[\![M]\!]\ \rho\backslash x\ )^\tau$

$\mathcal{R}[\![\textbf{let}\ x^\sigma = M_1\ \textbf{in}\ M_2)^\tau]\!]\ \rho =$ *if* σ is overloaded *then* $\mathcal{R}[\![M_2]\!]\ \rho\{\ \mathcal{R}[\![M_1]\!]\rho\ /x^\sigma\ \}$

else $(\ \textbf{let}\ x^\sigma = (\ \mathcal{R}[\![M_1]\!]\rho\ )\ \textbf{in}\ (\mathcal{R}[\![M_2]\!]\ \rho\backslash x\ )\ )^\tau$

$\mathcal{R}[\![\textbf{letop}\ x{:}\ \omega\ \textbf{in}\ M]\!]\ \rho = \mathcal{R}[\![M]\!]\ \rho$

$\mathcal{R}[\![M_1{:}\ \sigma\ \textbf{extends}\ x\ \textbf{in}\ M_2]\!]\ \rho = \mathcal{R}[\![M_2]\!]\ \rho\{\ \mathcal{R}[\![M_1]\!]\rho\ /x^\sigma\ \}$

$\mathcal{R}[\![M]\!]$ traverses the expression M, recursively expanding instances of overloaded operators until every letop- and extend-clause has been removed. Upon encountering $M_1 : \forall\alpha_i.\tau$ **extends** x **in** M_2, the resolution environment ρ is enhanced by the association $x^{\forall\alpha_i.\tau} \rightarrow \mathcal{R}[\![M_1{}^\tau]\!]\rho$. If, during the resolution of overloadings in M_2, some instance of this particular overloading of x is found, say $x^{S(\tau)}$, where S instantiates some overloaded type variable in τ to a type c(...), then x will be replaced by the result of resolving all overloadings in $S(\mathcal{R}[\![M_1]\!]\rho)$. let-clauses are also removed from M, if they introduce overloaded definitions. This corresponds to the view, that overloaded function definitions are not functions in the usual sense, but macros which are expanded according to an implicit type parameter.

Note, that for any well-typing $\langle O,A\rangle \vdash M : \tau$, $\mathcal{R}[\![M^\tau]\!]\ \rho$ is well defined if $x{:}\sigma \in A \Rightarrow \mathcal{R}[\![x^{\tau'}]\!]\rho$ well defined for every generic instance $\tau' <_O \sigma$.

Theorem: *semantic soundness of* $\mathcal{R}$*:*

Let $\langle O,A\rangle \vdash M : \tau$ be a well-typing, $\eta \in$ **Env**, $\varphi \in$ **Tenv**, $\rho \in$ **Renv** and $M' = \mathcal{R}[\![M^\tau]\!]\rho$.

If $x{:}\sigma \in A \Rightarrow \mathcal{E}[\![\ \mathcal{R}[\![x^{\tau'}]\!]\rho\ ]\!]\eta \in \mathcal{T}_O[\![\tau']\!]\varphi$ for every $\tau' <_O \sigma$ then $\mathcal{E}[\![M']\!]\eta \in \mathcal{T}_O[\![\overline{A\tau}\ ]\!]\varphi$.

4.3 Runtime Overloading Resolution

As an alternative to static overloading resolution we present a semantics for dynamic, i.e. runtime overloading resolution. The key idea behind this scheme is, that given a well-typed application of an overloaded operator, one can determine the particular overloading instance that is needed to compute the result by just looking at the summands of arguments.

Speaking in operational terms this implies, that arguments of overloaded operators have to be evaluated before applying the operator, thereby making it strict in its overloaded argument positions! An additional semantic complication is due to the fact, that the meaning of overloaded operators cannot be fixed statically in the declaring scope: Suppose we define an overloading for set-equality to be used in M, reducing equality on sets to equality on set elements. Suppose further, that we define a list comparison overloading inside M to be used in some yet deeper nested expression M'. Then we would certainly expect sets of lists of integers to be comparable in M'. However, having fixed the meaning of set equality to overloadings visible in M we can only compare lists of sets.

A similar observation can be made in the simpler case of let-introduced overloaded functions, which leads us to the introduction of a runtime equivalent of the static overloading resolution environment called operator environment.

$$\textbf{Openv} = \textbf{Id} \twoheadrightarrow \textbf{C} \twoheadrightarrow \textbf{Openv} \rightarrow \textbf{V}$$

Intuitively, $\vartheta \in$ **Openv** maps operator identifiers to functions which, when given the outermost type constructor (or summand) of the arguments of an operator application, will return a function mapping an operator environment to a real function (seen as an element of **V**).

Moreover, let-introduced identifiers will be treated as elements of the domain **Openv** $\rightarrow$ **V**, requiring

a change to the static environment domain.

$\mathbf{Env'} = \mathbf{Id} \rightarrow (\mathbf{V} \oplus (\mathbf{Openv} \rightarrow \mathbf{V}))$

Finally, the semantics is given by the function $\mathcal{E}'$: **Expr** → **Env'** → **Openv** → **V**, assuming that identifiers introduced in let- and extend clauses have been annotated with their type rsp. overloading scheme by the type inference algorithm:

$\mathcal{E}'[\![x]\!]\ \eta\ \vartheta$ = *if* $\eta(x) \in$ **Openv** → **V** *then* $\eta(x)\ \vartheta$ *elsif* $\eta(x) \in$ **V** *then* $\eta(x)$ *else wrong*

$\mathcal{E}'[\![\lambda x.M]\!]\ \eta\ \vartheta = (\lambda v.\ \mathcal{E}'[\![M]\!]\ \eta\{\ v/x\ \}\ \vartheta)$

$\mathcal{E}'[\![M_1\ M_2]\!]\ \eta\ \vartheta$ = *if* $f \in F$ *then* $(f|F)\ v$ *else wrong* , where $f = \mathcal{E}'[\![M_1]\!]\eta\vartheta$, $v = \mathcal{E}'[\![M_2]\!]\eta\vartheta$

$\mathcal{E}'[\![\mathbf{let}\ x^{\sigma} = M_1\ \mathbf{in}\ M_2]\!]\ \eta\ \vartheta$ = *if* σ is overloaded *then* $\mathcal{E}'[\![M_2]\!]\ \eta\{\ \mathcal{E}'[\![M_1]\!]\eta\ /x\ \}\ \vartheta$
else $\mathcal{E}'[\![M_2]\!]\ \eta\{\ \mathcal{E}'[\![M_1]\!]\eta\ /x\ \}\ \vartheta$

$\mathcal{E}'[\![\mathbf{letop}\ x\colon \omega\ \mathbf{in}\ M]\!]\ \eta\ \vartheta = \mathcal{E}'[\![M]\!]\ \eta\{\ \mathrm{resolve}(x,\omega)/x\ \}\ \vartheta\{\ \lambda c.\lambda\vartheta'.\ \perp\ /x\ \}$

$\mathcal{E}'[\![M_1\colon \sigma\ \mathbf{extends}\ x^{\omega}\ \mathbf{in}\ M_2]\!]\ \eta\ \vartheta =$
if $\eta(x)$ **Openv** → **V** and σ overloads ω with c(...)
then $\mathcal{E}'[\![M_2]\!]\ \eta\ \vartheta\{\ (\ (\vartheta\ x)\{\ \mathcal{E}'[\![M_1]\!]\eta\ /x\ \}\ )\ /\ x\ \}$ *else wrong*

resolve: **(O, Id)** → **Openv** → **V**

resolve $(\omega_1 \times \ldots \times \omega_n \rightarrow \omega_r\ ,\ x)\ \vartheta =$
$\lambda(v_1,\ldots,v_n).$ *if* $\exists\ c \in C.\ \omega_i = \$ \Rightarrow v_i \in S_c(V^{a(c)})$
then $(\ (\vartheta\ x)\ c\ \vartheta)\ (v_1,\ldots,v_n)$
else wrong

Rather than delving into the details of this denotational semantics, we give a theorem stating the relationship between compile time and runtime overloading resolution, namely that runtime overloading resolution delivers the same results, apart from possibly introducing nontermination.

Theorem: $\mathcal{E}'$ *weakly implements* $\mathcal{E} \cdot \mathcal{R}$

Let $\langle O,A\rangle \vdash M^{\tau}$ be a well typing , $\eta' \in$ **Env'**, $\vartheta \in$ **Openv**, $\eta \in$ **Env** and $\rho \in$ **Renv**.
If $x:\sigma \in A \Rightarrow \mathcal{E}'[\![x]\!]\eta'\ \vartheta \sqsubseteq \mathcal{E}[\![\ \mathcal{R}[\![x^{\tau'}]\!]\rho\ ]\!]\eta$ for every $\tau' <_o \sigma$
then $\mathcal{E}'[\![M]\!]\eta'\ \vartheta \sqsubseteq \mathcal{E}[\![\ \mathcal{R}[\![\ M^{\tau}]\!]\ ]\!]\ \eta$.

5 Final Remarks

We have shown that the restriction to parametric overloading results in rather simple and efficient tpye-inference algorithms, while still allowing the specification of many useful overloaded functions. Moreover, it turns out, that overloading resolution is possible either statically, thus yielding no runtime overhead at all, or dynamically (at runtime), which is particularly advantagous in the context of a language incorporating a module concept. Indeed, any sensible mixture of the two strategies can be used in a specific implementation.

On the other hand, one may object that the restriction imposed on the set of possible overloadings is to severe, disallowing some useful overloaded operations. Although we do not share this opinion, we suppose that it is possible to integrate unrestricted with parametric overloading, using the framework of *context relations* described in [BaSne86].

We have successfully used parametric overloading in developing the predefined operations of the functional programming language SAMPLE (see [JGK87] for some details of the type system). Having used the SAMPLE environment for over a year in a number of projects, our personal experience

shows that the inclusion of parametric overloading has significantly improved the usability and type security of SAMPLE. Our next step will therefore be the integration of user definable overloading for operations on abstract data types, along the lines outlined in chapter 4.2 and 4.3.

A number of possible extensions to parametric overloading have not been discussed here, partly due to space limitations, partly because their inclusion would have over complicated the presentation. First, it is possible to combine parametric overloading with the subtyping displicines of [Mitchell84], [Letsch86] and [FuhMi87] (we have actually implemented a type inference algorithm handling both concepts for the SAMPLE language). Second, the set of possible overloading schemes can be extended to cope with overloadings such as $\forall\alpha.\ \alpha \times \$(\alpha) \rightarrow bool$, where $ can appear as a typeconstructor. A main application of this kind of overloading is the member function, with the typeset { $\forall\alpha.\alpha \times list(\alpha) \rightarrow bool$, $\forall\alpha.\alpha \times set(\alpha) \rightarrow bool$, ... }. Third, one can devise a scheme which removes the strictness restriction from runtime overloading resolution by implicitly adding type parameters to overloaded functions.

References:

[ASU86] A.V. Aho, R. Sethi and J.D. Ullman: *Compilers: Principles, Techniques, and Tools*, p384, 1986.

[BaSne86] R. Bahlke and G. Snelting: *The PSG-System: From Formal Language Definitions to Interactive Programming Environments*, TOPLAS 8,4, p547-576, October 1986.

[BMQS80] R. Burstall, D.B. MacQueen and D. Sanella: *HOPE: An Experimental Applicative Language*, 1st International LISP Conference, Stanford 1986.

[Coppo80] M. Coppo: *An Extended Polymorphic Type System for Applicative Languages*, LNCS 88, p194-204, September 1980.

[DaMi82] L. Damas and R.Milner: *Principal Type Schemes for Functional Programs*, IX POPL, p207, January 1982.

[CDDK86] D. Clement, J. Despeyroux, T. Despeyroux and G. Kahn: *A simple applicative language: Mini-ML*, 1986 ACM Symposium on LISP and Functional Programming, p13-27, 1986.

[FuhMi87] Y. Fuh and P. Mishra: *Type Inference With Subtypes*, Manuscript, SUNY at Stony Brook, July 1987.

[JGK87] M. Jäger, M. Gloger and S. Kaes: *SAMPLE - A Functional Language*, Report PI-R5/87, TH-Darmstadt, Fachbereich Informatik.

[Kaes87] S. Kaes: *Parametric Overloading in Polymorphic Programming Languages*, Report PI-R7/87, TH-Darmstadt, Fachbereich Informatik.

[Letsch86] T. Letschert: *Typinferenzsysteme*, Doctoral Thesis, TH Darmstadt, Fachbereich Informatik, 1986.

[Milner78] R. Milner: *A Theory of Type Polymorphism in Programming*, JCCS 17,3, p348-375, 1978.

[Milner84] R. Milner: *A Proposal for Standard ML*, 1984 ACM Symposium on LISP and Functional Programming, p184-197, Austin, August 1984.

[Mitchell84] J.C. Mitchell: *Coercion and Type Inference*, XI POPL, p175-185, 1984.

[MQPS84] D.B. MacQueen, G.D. Plotkin and R. Sethi: *An Ideal Model for Recursive Polymorphic Types*, XI POPL, p165-174, 1984.

[Str67] C. Strachey: *Fundamental Concepts in Programming Languages*, International Summer School in Computer Programming, Kopenhagen 1967.

[Turner85] D.A. Turner: *Miranda: A non-strict Functional Language with Polymorphic Types*, LNCS 201, September 1985.