Elemental Function Overloading in Explicitly Typed Languages

Claes Thornberg[1] and Björn Lisper[2]

[1] Department of Teleinformatics
Royal Institute of Technology
Stockholm, Sweden
`claest@it.kth.se`

[2] Department of Computer Engineering
Mälardalen University
Västerås, Sweden
`bjorn.lisper@mdh.se`

Abstract. Elemental intrinsic overloading is used successfully in vector parallel and data parallel programming languages. It allows scalar operators to be applied to arguments of array type, where the semantics is a new array where every element is the result of applying the operator to the corresponding elements of the argument array. This type of overloading makes programs with extensive use of array operations easier to read, write and maintain. However, it is typically restricted to allow overloading only of built-in functions and defined only for operators on one parallel data type, mostly arrays. In order to extend this feature to a larger class of languages, we propose *elemental function overloading* together with a polymorphic type discipline. Elemental function overloading, which is defined for all functions, also incorporates *promotion* of "scalar" values. In this paper we formalize the concept of elemental functional overloading in an explicitly typed polymorphic language and present an algorithm for resolving this overloading. We also make a preliminary study of elemental function overloading in an implicitly typed polymorphic language with type inference.

1 Introduction

Array-based languages like Apl [13], Fortran 90 [1] and HPF [11] and systems like MatLab [18] allow *elemental intrinsic overloading*, where a scalar operator may be applied to arguments of array type. The semantics is that a new array is produced where every element is the result of the operator applied to the corresponding elements of the argument arrays. Apl, Fortran 90, and HPF also allow *vector valued subscripts* or *indirect indexing*, where an array with index values is given as index to another array. The semantics is a new array, of the same length as the argument array, which holds the indirectly indexed elements of the outer array. The following Fortran 90 code exemplifies the use of these features:

```
REAL, DIMENSION (N) :: A, B, C
INTEGER, DIMENSION (N) :: IND
REAL, DIMENSION (M) :: D
C = SQRT(A) + B**2 + D(IND)
```

After the operation, every element `C(I)` will hold the value of the expression `SQRT(A(I)) + B(I)**2 + D(IND(I))`. This style of programming, which goes back to APL, is appreciated among scientific computing programmers since it yields a succinct, yet natural and readable way for expressing many algorithms [14, p. 56].

A similar, but slightly different kind of overloading was provided in late versions of the data parallel language *lisp [21] for the Connection Machine. An example is the expression

```
(+!! x!! 2)
```

Here, `x!!` is a *pvar*, an array distributed throughout the Connection Machine, and `+!!` is an operation for elementwise addition of pvar's. Thus, the second argument ought to be a pvar as well, but the overloading allows the scalar 2 to be interpreted as a pvar with elements 2 throughout. This overloading is sometimes called *promotion*.

However, existing languages with these features impose a number of restrictions on their use. First, Fortran 90 and HPF allow the use of elemental intrinsic overloading only for a number of builtin operations, not for functions in general. [1]. Second, these features are usually only allowed for a single indexed data type, like arrays. It is not hard to see that they would work for indexed data types in general. Elemental overloading and promotion, in particular, should also be possible to apply for other structured types like homogeneous lists. Third, these features only appear in explicitly typed monomorphic languages. Moreover, using this kind of overloading in languages with nested structures may lead to ambiguities [19].

The aim of the work reported here is to give these kinds of overloading a firm theoretical basis, in order to overcome the somewhat ad hoc nature of current approaches and to make the features more generally applicable. In particular we want to support the generalization from intrinsic operations to functions in general, from certain indexed data types to homogeneous indexed datatypes in general, from monomorphism to polymorphism, and, further down the road, from explicitly typed languages to languages with Hindley-Milner style type inference [12, 16, 4, 3]. For this purpose, we have formalized the notion of elemental function overloading, extending the Hindley-Milner type system for an explicitly typed functional language with a rewrite procedure which transforms overloaded terms into well-typed terms where the overloading is resolved.

[1] For MatLab the overloading works also for user-defined functions. This is since Matlab has a degenerate type system where all data are matrices. Thus, operators and functions are not really overloaded. This approach works only if the language has a single data type for all data.

We prove that this system does not rewrite terms which are typed by the original type system, and that successfully rewritten terms always have a type in the original system. Furthermore, if a term is not rewritten, it is assigned the same type in both the original and our proposed system. We call these properties *soundness* and *transparency* with respect to the original system, respectively. For this explicitly typed system we also give an algorithm which resolves the overloading for overloaded terms and calculates the type of the result. We prove that the algorithm is sound and complete with respect to the extended type system. Furthermore, we have adapted the formalization of elemental function overloading to an implicitly typed language. We conclude by showing that the properties that hold for the explicitly typed system, also holds for the implicitly typed system. However, we have not yet developed an algorithm for inferring type and resolving elemental function overloading in the implicitly typed language.

Due to space constraints, full proofs of all properties are left out. However, they can be found in [22].

2 Related Work

Although elemental intrinsic overloading, vector valued subscripts, promotion and other collection-oriented features [19] have been proposed for and used in many different programming languages, most significantly, maybe, in array- and data parallel languages, only a few have worked on the formal definition of these concepts and how to resolve them. For instance, the HPF specification [11] does not give any formal account for how to resolve these forms of overloading even though the language supports them. In APL and APL2, the semantics of implicit scaling is both complex and *ad hoc*. Our work, on the other hand gives a formal definition of the combination of three features elemental function overloading, vector valued subscripts, and promotion, and how to resolve these kinds of overloading.

Elemental overloading of operators to functions over time has been proposed for the Duration Calculus [2], and in a proposed extension to the Z specification language [5, 7]. A complete algorithm for this scheme has been presented [6], but the algorithm is reduced to a proliferation of case-sensitive lifting processes [5, p. 11]. In contrast, our algorithm is quite easy to understand, comprising only one rule schema, which describes how an expression is transformed.

Thatte [20] proposes a type system for implicit scaling based on structural subtyping, defined for a monomorphic language which is a simply typed dialect of the λ-calculus with lists and pairs. Thatte's approach has several drawbacks. First, it does not incorporate explicit parametric polymorphism. Second, Thatte's system also restrict the notion of what a scalar value is. In comparison, our approach incorporates parametric polymorphism, where explicit type instantiations resolve the ambiguities that may otherwise arise when combining scaling and polymorphism, see Example 3. Also, our proposed system extends the notion of what a scalar value is.

The resolution of class-based overloading in Haskell [8] and related systems of overloading [17] use combined type inference and rewrite systems somewhat reminiscent of ours. However, there are several important differences between this and our proposed approach, among which these are the most important.

- It requires a language with type classes or some similar mechanism.
- Overloading in Haskell works on class basis, and an instance declaration has to be made for each class, for which we want the overloading mechanism.
- Using Haskell's type classes to resolve this overloading is potentially costly. It will normally result in dictionaries which are carried around and used in run-time, while in our approach, all overloading is resolved statically.

The original motivation for this work comes from the second author's work on abstract modelling of indexed entities as partial functions [9, 15]. Our results are directly applicable in this context.

3 Elemental Function Overloading

Our type inference and rewrite system resolves elemental intrinsics overloading, but for functions in general, not just a predefined set of operators. Thus, we call it *elemental function overloading.* The system also performs promotion of "scalars". Some examples demonstrate what kinds of transformations the system performs (in all examples, introduced variables are fresh):

Example 1. Elemental intrinsic overloading: consider the expression

$$f(a_1, a_2)$$

where the types for f, a_1 and a_2 are

$$\begin{aligned} f &: \tau_1 \times \tau_2 \to \tau \\ a_1 &: \iota \to \tau_1 \\ a_2 &: \iota \to \tau_2 \end{aligned}$$

The "elemental intrinsics" interpretation is to see this as an elementwise application of f to a_1 and a_2. When a_1 and a_2 are functions, the intended interpretation is

$$\lambda x.f(a_1\, x, a_2\, x) : \iota \to \tau,$$

the function which returns $f(a_1\, x, a_2\, x)$ for each "index" x. Our system will indeed rewrite $f(a_1, a_2)$ in this way, given that f, a_1 and a_2 have the types above inferred.

In the example above, there are two important things to note. First, the types of the domains of the two actual parameters are the same, in this case ι, while the type of the range of each actual parameter is the same as the type of the corresponding formal parameter. Second, the type of the transformed function application is a function, in this case from ι, the type of the domains of the actual parameters, to τ, the type of the range of the function f.

Example 2. Elemental overloading on nested structures: reconsider

$$f(a_1, a_2)$$

but where the types for f, a_1 and a_2 now are

$$\begin{aligned} f &: \tau_1 \times \tau_2 \to \tau \\ a_1 &: \iota_1 \to \iota_2 \to \tau_1 \\ a_2 &: \iota_1 \to \iota_2 \to \tau_2 . \end{aligned}$$

Our system will now transform $f(a_1, a_2)$ into

$$\lambda x_1 \lambda x_2 . f(a_1\, x_1 x_2, a_2\, x_1 x_2) : \iota_1 \to \iota_2 \to \tau$$

The functions a_1 and a_2, with their curried function types, model nested data structures such as arrays of arrays or nested lists. A moment of thought shows that the resulting function models elementwise application on nested data structures. (For each possible pair of indices, first index the "outermost" structure, then the "innermost", then apply f.)

Example 3. Polymorphism cannot give rise to ambiguities in our system when combining it with elemental overloading on nested structures. Consider the expression

$$\mathit{reverse}\, f$$

where f models some nested structure, such as arrays of arrays, and *reverse* is a polymorphic function. Assuming that their types are

$$\begin{aligned} &f : \mathit{int} \to \mathit{int} \to \tau \\ &\mathit{reverse} : \forall \alpha . (\mathit{int} \to \alpha) \to \mathit{int} \to \alpha \end{aligned}$$

the expression *reverse* f has two plausible interpretations. We can either apply reverse on the outer structure, i.e. we instantiate α above to $\mathit{int} \to \tau$. In this case, no translation of *reverse* f takes place. This is the natural interpretation with polymorphic functions. However, by instantiating α to *int*, the interpretation is changed. Now *reverse* will be applied to the inner structure, i.e. the expression *reverse* f is translated to

$$\lambda x . \mathit{reverse}(fx) .$$

In Thatte's system, these kinds of ambiguities are avoided by restricting the language to a monomorphic one, thereby automatically selecting the latter interpretation over the former. However, by using polymorphism with explicit instantiations, we can let the user choose either of these interpretations. In Core-XML, see Sect. 5, we could write either

$$\mathit{reverse}[\mathit{int} \to \tau]\, f$$

which would apply *reverse* on the outer structure, or we could write

$$\mathit{reverse}[\tau]\, f$$

which in our proposed system would be transformed to

$$\lambda x.\mathit{reverse}[\tau]\,(f\,x)$$

i.e. applying *reverse* on the inner structure of f.

Example 4. Promotion: again, reconsider

$$f(a_1, a_2)$$

where the types for the subexpressions now are

$$\begin{aligned} &f : (\iota \to \tau_1) \times (\iota \to \tau_2) \to \tau \\ &a_1 : \iota \to \tau_1 \\ &a_2 : \tau_2. \end{aligned}$$

Our system now transforms $f(a_1, a_2)$ into

$$f(a_1, \lambda x.a_2) : \tau$$

This can be seen as promotion of the "scalar" a_2 into the constant function $\lambda x.a_2$.

4 Types and Notation

Throughout the paper, we let τ denote types, α and β type variables, and σ type schemes. In addition we have type constructors $\to$, denoting function types, and an infinite set of constructors for creating tuple types. Types and type schemes are defined recursively as

$$\begin{array}{rll} \tau ::= & \tau \to \tau & \text{function types} \\ | & (\tau, \ldots, \tau) & \text{tuples} \\ | & \alpha & \text{type variables} \\ | & b & \text{predefined basic types.} \\ \sigma ::= & \tau & \\ | & \forall \alpha.\sigma & \end{array}$$

Type variables that are not quantified by a type scheme σ are free in σ.

For the inference systems in this paper we use the following notation.

A	a set containing at most one assumption about each identifier x,
$x : \tau$	an assumption associating the program variable x with type τ,
$FV(A)$	all free variables in the set of assumptions,
A_x	the result of removing any assumption about x from A,
$A_{\bar{\imath}}$	the result of removing the assumptions about the variables in the sequence $\bar{\imath}$,

$A \cup \{x : \tau\}$	a set of assumptions, where the type for x is τ,
$A \vdash_{\mathrm{HM}} e : \sigma$	from the set of assumptions A, we can infer that e has the type σ in the original Hindley-Milner system,
$A \vdash e \leadsto e' : \sigma$	from the set of assumptions A, we are allowed to rewrite e to e', where e' has the type σ,
$\sigma[\alpha := \tau]$	the result of substituting τ for all free occurrences of α in σ,
$\dfrac{A}{B}$	B can be inferred from A.

5 Core-XML

For the purpose of explaining and formalizing our ideas we use a dialect of the simple language Core-XML [10], which is an explicitly typed counterpart to Core-ML, the language used in many other presentations of type inference systems, e.g. in the works by Damas and Milner [4] and Cardelli [3]. The dialect we use is slightly less explicitly typed than the original and has construction and deconstruction of tuples, since we make explicit use of these constructs later when defining the transformation rules of our system.

Expressions in this version of Core-XML are defined recursively as

$e ::= x$	variable
$\mid\ \lambda x : \tau . e$	abstraction
$\mid\ ee$	application
$\mid\ \textbf{let}\ x = e\ \textbf{in}\ e$	definition
$\mid\ (e_1, \ldots, e_n)$	n-ary tuple
$\mid\ \pi_i e$	projection
$\mid\ \Lambda\alpha . e$	type abstraction
$\mid\ e[\tau]$	type application.

As can be seen, Core-XML provides explicit type information for every variable introduced with λ-abstractions. A type inference system for Core-XML, adapted from [10], is given in Fig. 1. The inference system is simple, and in principle, inferring the type of an expression boils down to type-checking the expression. In the inference system, recursive function definitions are not allowed, instead the predefined polymorphic fixed-point operator fix : $\forall\alpha.(\alpha \to \alpha) \to \alpha$ can be used.

6 Elemental Function Overloading in Core-XML

It is quite straightforward to incorporate and formalize the transformations, informally described in Sect. 3, into the system for typing Core-XML expressions. First the rules of the ordinary system are adapted to reflect the new syntax of judgements. As an example, the rule for function application

$$\frac{A \vdash_{\mathrm{HM}} e : \tau_a \to \tau \quad A \vdash_{\mathrm{HM}} e_a : \tau_a}{A \vdash_{\mathrm{HM}} e e_a : \tau}$$

$$A_x \cup \{x : \sigma\} \vdash_{\mathrm{HM}} x : \sigma$$

$$\frac{A_x \cup \{x : \tau_a\} \vdash_{\mathrm{HM}} e : \tau}{A \vdash_{\mathrm{HM}} \lambda x : \tau_a.e : \tau_a \to \tau}$$

$$\frac{A \vdash_{\mathrm{HM}} e : \tau_a \to \tau \quad A \vdash_{\mathrm{HM}} e_a : \tau_a}{A \vdash_{\mathrm{HM}} ee_a : \tau}$$

$$\frac{A \vdash_{\mathrm{HM}} e_x : \sigma \quad A_x \cup \{x : \sigma\} \vdash_{\mathrm{HM}} e : \tau}{A \vdash_{\mathrm{HM}} \mathbf{let}\ x = e_x\ \mathbf{in}\ e : \tau}$$

$$\frac{A \vdash_{\mathrm{HM}} e_1 : \tau_1 \ \cdots \ A \vdash_{\mathrm{HM}} e_n : \tau_n}{A \vdash_{\mathrm{HM}} (e_1, \ldots, e_n) : \tau_1 \times \cdots \times \tau_n}$$

$$\frac{A \vdash_{\mathrm{HM}} e : \tau_1 \times \cdots \times \tau_n}{A \vdash_{\mathrm{HM}} \pi_i e : \tau_i} \quad 1 \leq i \leq n$$

$$\frac{A \vdash_{\mathrm{HM}} e : \sigma}{A \vdash_{\mathrm{HM}} \Lambda\alpha.e : \forall\alpha.\sigma} \quad \alpha \notin FV(A)$$

$$\frac{A \vdash_{\mathrm{HM}} e : \forall\alpha.\sigma}{A \vdash_{\mathrm{HM}} e[\tau] : \sigma[\alpha := \tau]}$$

Fig. 1. The Core-XML type system without elemental function overloading

is changed to

$$\frac{A \vdash_{\mathrm{HM}} e \rightsquigarrow e' : \tau_a \to \tau \quad A \vdash_{\mathrm{HM}} e_a \rightsquigarrow e'_a : \tau_a}{A \vdash_{\mathrm{HM}} ee_a \rightsquigarrow e'e'_a : \tau}$$

Then the transformations from Sect. 3 are formalized, see Fig. 2. The transformations rules are described in the form of a schema since the number of transformation rules is infinite. Figures 1, with judgements modified according to the above, and 2 together define our type and rewrite system for elemental function overloading in Core-XML.

6.1 Informal Description of Transformation Rules

The transformation rule schema in Fig. 2 is quite detailed. Therefore, we give an intuitive understanding of how it is supposed to work. Let us first make the following definition:

Definition 1. $\succeq$ *("larger or equal than") is a relation on types defined by*

- $\tau \succeq \tau$
- $\iota \to \tau \succeq \tau'$ *iff* $\tau \succeq \tau'$

Let

$$A_{\bar{x}} \vdash e \leadsto e' : \tau_1 \times \cdots \times \tau_n \to \tau_r$$
$$A_{\bar{x}} \vdash e_a \leadsto e'_a : \gamma_1 \times \cdots \times \gamma_n$$

be the premises in the rule. For $i \in \{1, \ldots, n\}$ determine k_i, I_l, I_g and I_e such that $\{I_l, I_g, I_e\}$ is a partition of $\{1, \ldots, n\}$ and the following relations between the different γ_i and τ_i all hold.

$$\begin{array}{ll} \tau_i \equiv \iota_{i,k_i} \to \cdots \to \iota_{i,1} \to \gamma_i & i \in I_l \\ \gamma_i \equiv \iota_{k_i} \to \cdots \to \iota_1 \to \tau_i & i \in I_g \\ \tau_i \equiv \gamma_i & i \in I_e \end{array}$$

Then define

$$\begin{array}{rll} k_{\max} &= \max_{i \in I_g} k_i & I_g \neq \emptyset \\ k_{\max} &= 0 & I_g = \emptyset \\ y_i &= \pi_i e'_a & i \in I_e \\ y_i &= \lambda x_{i,k_i} : \iota_{i,k_i} \ldots \lambda x_{i,1} : \iota_{i,1}.\pi_i e'_a & i \in I_l \\ y_i &= (\pi_i e'_a)\, x_{k_i} \cdots x_1 & i \in I_g \end{array}$$

where all x_k and $x_{i,k}$ are fresh variables. More specifically, they belong to the sequence $\bar{x}$ and do not belong to $\mathrm{Dom}(A_{\bar{x}})$. The schema for the transformation rules is now given by

$$\frac{\begin{array}{c} A_{\bar{x}} \vdash e \leadsto e' : \tau_1 \times \cdots \times \tau_n \to \tau_r \\ A_{\bar{x}} \vdash e_a \leadsto e'_a : \gamma_1 \times \cdots \times \gamma_n \end{array}}{\begin{array}{c} A \vdash e e_a \leadsto \lambda x_{k_{\max}} : \iota_{i,k_i} \ldots \lambda x_1 : \iota_{i,1}.e'(y_1, \ldots, y_n) : \\ \iota_{k_{\max}} \to \cdots \to \iota_1 \to \tau_r, \text{ when } k_{\max} > 0 \\ \tau_r, \text{ otherwise} \end{array}} [\mathit{TRANS}]$$

Fig. 2. Schema for transformation rules

Now assume we have an expression $f(a_1, \ldots, a_n)$, where the types for the subexpressions, under the given assumptions, are $f : \tau_1 \times \cdots \times \tau_n \to \tau$ and $a_i : \gamma_i$. In short, what the transformation rule $[\mathit{TRANS}]$ does can be explained with the following two steps.

1. For each argument a_i: if $\tau_i \succeq \gamma_i$, then λ-abstract over a_i on fresh variables, so that the new type for a_i equals τ_i (promotion). Otherwise, check that $\gamma_i \succeq \tau_i$, and that the γ_i for these i all are comparable using $\succeq$ when the τ_i's are replaced with a common type τ.
2. If the previous step was successful, then λ-abstract the whole expression on fresh variables, and apply each argument a_i where $\gamma_i \succeq \tau_i$ to sufficiently many of these variables so the resulting argument has type τ_i.

The actual rule schema for our system (Fig. 2) is of course a bit more complex since it summarizes all possible rules, but in essence it follows the above procedure.

6.2 A Digression

The definition of the rule schema requires some justification. First, the relation between the types of the formal and actual parameter is defined on the basis that we want to overload homogeneous aggregates of data modelled as functions. Therefore, we require that an element of the actual argument is of the correct type, or is a functional value which can be used to get a value of the proper type, or is a value which can be transformed to a value of the expected functional type. For instance, a function f of type $\tau_1 \to \tau_2 \to \tau_3$, can through function application be used to acquire values of types $\tau_2 \to \tau_3$ and τ_3. A value of type τ_3 can through λ-abstractions be transformed to values of types $\tau_2 \to \tau_3$ and $\tau_1 \to \tau_2 \to \tau_3$. Second, it must be noted that elemental function overloading is defined only on the first argument of a function.

6.3 Some properties

For our type system with elemental function overloading, the following two important properties hold. We call them *transparency* and *soundness*, respectively.

Proposition 1 (Transparency). $\forall A \forall e \forall \sigma . A \vdash_{\mathrm{HM}} e : \sigma \Rightarrow A \vdash e \rightsquigarrow e : \sigma$ *holds, i.e. an expression which can be given a type without being subject to any transformation will not be transformed.*

Proof. By induction over derivations. It is easy to see that whenever a typing rule in the original system is used, we can use the corresponding rule in the combined system. And if we do this, no transformation of e will take place. □

Proposition 2 (Soundness). $\forall A \forall e \forall e' \forall \sigma . A \vdash e \rightsquigarrow e' : \sigma \Rightarrow A \vdash_{\mathrm{HM}} e' : \sigma$ *holds, i.e. the type given to a transformed expression is the same as the original type system would assign to it.*

Proof. By induction over derivations. All cases are quite straightforward, except when a transformation has taken place. But in that case we notice that the type given to the transformed expression always is correct according to the types of the (transformed) subexpressions. □

6.4 An Algorithm for Resolving Overloading in Core-XML

We now give an algorithm for type assignment and rewriting according to our type system in Figs. 1 and 2. The algorithm is outlined in Fig. 3, where only the part actually transforming an expression is given in detail. As input the algorithm takes a set of assumptions, and a program for which a type should be inferred. As output it produces a transformation of the input program and a type for the transformed program. For all cases except when there is a type mismatch, the algorithm just checks that types are used appropriately. When there is a type mismatch, the algorithm tries to match one of the transformation rules. If it succeeds, the transformed expression and its type is returned, otherwise an error occurs.

```
Input: A set of assumptions $A$ and a program $e$.
Output: A pair $(e', \sigma)$, the meaning and type of $e$ under $A$

$\mathrm{Tr}(A,x) = \ldots$

$\mathrm{Tr}(A,\ \Lambda\alpha.e) = \ldots$

$\mathrm{Tr}(A,\ e[\tau]) = \ldots$

$\mathrm{Tr}(A,\ \lambda x : \tau_a.e) = \ldots$

$\mathrm{Tr}(A,\ \mathit{let}\ x = e_x\ \mathit{in}\ e) = \ldots$

$\mathrm{Tr}(A,\ ee_a) =$
   let $(e', \tau) = \mathrm{Tr}(A,e);\ (e'_a, \tau_a) = \mathrm{Tr}(A,\ e_a)$ in
   case $\tau$ of
     $\tau_a \to \tau_r$ then $(e'e'_a, \tau_r)$
     $\tau_1 \times \cdots \times \tau_n \to \tau_r$ then
     case $\tau_a$ of
       $\gamma_1 \times \cdots \times \gamma_n$ then
         let forall $i \in \{1, \ldots, n\}$ $y_i =$
                 $(\pi_i e'_a)x_{n_i} \ldots x_1$ when $\gamma_i \equiv \iota_{n_i} \to \cdots \to \iota_1 \to \tau_i$
                 $(\pi_i e'_a)$ when $\gamma_i \equiv \tau_i$
                 $\lambda x_{i,n_i} \ldots \lambda x_{i,1}.(\pi_i e'_a)$ when $\tau_i \equiv \iota_{i,n_i} \to \cdots \to \iota_{i,1} \to \gamma_i$
                 otherwise undefined
             $I_g = \{l | \gamma_l \equiv \iota_{n_l} \to \cdots \to \iota_1 \to \tau_l\}$
             $n_{\max} =$ if $I_g = \emptyset$ then $0$ else $\max_{i \in I_g} n_i$ in
         if any $y_{i \in \{1,\ldots,n\}} =$ undefined then Fail
         else if $n_{\max} = 0$ then $(e'(y_1, \ldots, y_n), \tau_r)$
         else $(\lambda x_{n_{\max}} \ldots \lambda x_1.e'(y_1, \ldots, y_n), \iota_{n_{\max}} \to \cdots \to \iota_1 \to \tau_r)$
       else Fail
     else Fail
```

Fig. 3. Outline of algorithm for resolving elemental function overloading

The algorithm has the following properties:

Proposition 3 (Soundness). $\forall A \forall e \forall e' \forall \sigma . \mathrm{Tr}(A,e) = (e'\sigma) \Rightarrow A \vdash e \rightsquigarrow e' : \sigma$ *holds, i.e if the algorithm rewrites the program e to e' and gives a type σ for e', then it is possible to infer that transformation and type for the program.*

Proof. By induction over programs. Let the induction hypothesis be $\mathrm{Tr}(A,e) = (e', \sigma) \Rightarrow A \vdash e \rightsquigarrow e' : \sigma$ for all programs of length less than or equal to n. The base case where e is a variable naturally holds. By considering how programs are built, it is easy to show that the proposition always holds. □

Proposition 4 (Completeness). $\forall A \forall e \forall e' \forall \sigma . A \vdash e \rightsquigarrow e' : \sigma \Rightarrow \mathrm{Tr}(A, e) = (e', \sigma)$ *holds, i.e. if a transformation e' and type σ can be inferred for a program e then the algorithm will give that transformation and type for the program.*

Proof. By induction over derivations. □

7 Hindley-Milner Type Inference and Elemental Function Overloading

Our proposed system was originally intended for incorporating elemental function overloading in a polymorphic, implicitly typed language. The motivation for this, if any motivation is needed at all, is that elemental function overloading, polymorphism, and type inference have proven to be useful programming tools. Elemental function overloading makes programs with extensive use of array operations easier to read, write and maintain. Polymorphism can be used to parameterize code, making it both more readable and reusable, while type inference takes the burden of type annotation from the user. We believe that a combination of all these features in one language would also be useful.

For the purpose of a preliminary investigation of how well our system can be adapted to an implicitly typed language, we use Core-ML. To this language we have added construction and deconstruction of tuples since we make explicit use of these constructs later. The language is similar to Core-XML, see Sect. 5, but without explicit type applications and type abstractions. Also, λ-abstracted variables are not annotated.

A "classical" Hindley-Milner type inference system for our Core-ML dialect is given in Fig. 4. It is derived from the system in [4] by adding the rules for tuples and projections. We use this system as a reference to our own inference and rewrite system for Core-ML.

7.1 An Inference System for Elemental Function Overloading

It is quite straightforward to incorporate the transformations, informally described in Sect. 3, into a Hindley-Milner type inference system. First we modify the rules from the ordinary Hindley-Milner type system in Fig. 4, see Sect. 6. We then formalize the transformations in the same way as before, changing them slightly to allow for transformed subexpressions, see Fig. 5. The resulting transformation rule schema is almost identical to the schema presented in Fig. 2, except of course that the constructed λ-abstractions are not explicitly typed. Figures 4, with the modified judgements, and 5 together define our type inference and rewrite system for elemental function overloading in Core-ML.

For our implicitly typed system with elemental function overloading, the properties transpararency and soundness holds, see Properties 1 and 2. The proofs of these properties are similar to the corresponding proofs in the explicitly typed system.

Proposition 5 (Transparency). $\forall A \forall e \forall \sigma . A \vdash_{\mathrm{HM}} e : \sigma \Rightarrow A \vdash e \rightsquigarrow e : \sigma$

Proposition 6 (Soundness). $\forall A \forall e \forall e' \forall \sigma . A \vdash e \rightsquigarrow e' : \sigma \Rightarrow A \vdash_{\mathrm{HM}} e' : \sigma$

$$A_x \cup \{x : \sigma\} \vdash_{\mathrm{HM}} x : \sigma$$

$$\frac{A_x \cup \{x : \tau_a\} \vdash_{\mathrm{HM}} e : \tau}{A \vdash_{\mathrm{HM}} \lambda x.e : \tau_a \to \tau'}$$

$$\frac{A \vdash_{\mathrm{HM}} e : \tau_a \to \tau \quad A \vdash_{\mathrm{HM}} e_a : \tau_a}{A \vdash_{\mathrm{HM}} ee_a : \tau}$$

$$\frac{A \vdash_{\mathrm{HM}} e_x : \sigma \quad A_x \cup \{x : \sigma\} \vdash_{\mathrm{HM}} e : \tau}{A \vdash_{\mathrm{HM}} \mathbf{let}\ x = e_x\ \mathbf{in}\ e : \tau}$$

$$\frac{A \vdash_{\mathrm{HM}} e_1 : \tau_1 \quad \cdots \quad A \vdash_{\mathrm{HM}} e_n : \tau_n}{A \vdash_{\mathrm{HM}} (e_1, \ldots, e_n) : \tau_1 \times \cdots \times \tau_n}$$

$$\frac{A \vdash_{\mathrm{HM}} e : \tau_1 \times \cdots \times \tau_n}{A \vdash_{\mathrm{HM}} \pi_i e : \tau_i} \quad 1 \le i \le n$$

$$\frac{A \vdash_{\mathrm{HM}} e : \sigma}{A \vdash_{\mathrm{HM}} e : \forall \alpha.\sigma} \quad \alpha \notin FV(A)$$

$$\frac{A \vdash_{\mathrm{HM}} e : \forall \alpha.\sigma}{A \vdash_{\mathrm{HM}} e : \sigma[\alpha := \tau]}$$

Fig. 4. A Hindley-Milner type inference system with rules for tupling and projection

8 Conclusions

We have shown how the syntactical conveniences *elemental intrinsic overloading*, *vector valued subscripts*, and *promotion of scalars*, common in array- and data parallel languages such as Fortran 90 and *lisp, can be given a very general formulation in an abstract setting. Our results give these language constructs a firm theoretical basis, in the form of type systems and rewrite systems and soundness results for these. Our approach is also an improvement over earlier attempts in that we use a polymorphic language, allow overloading for both built-in and user-defined operations, and handle nested structures in a natural way.

We first developed a type assignment and rewrite system for the explicitly typed language Core-XML and presented an algorithm for resolving elemental function overloading in this setting. The algorithm provides a direct way to introduce the general forms of overloading we consider into explicitly typed languages. We then developed a type inference and rewrite system for the implicitly typed language Core-ML and showed that the system is sound and transparent with respect to the original Hindley-Milner system. This is a first step in an attempt

Let

$$\begin{array}{c} A_{\bar{x}} \vdash e \rightsquigarrow e' : \tau_1 \times \cdots \times \tau_n \to \tau_r \\ A_{\bar{x}} \vdash e_a \rightsquigarrow e'_a : \gamma_1 \times \cdots \times \gamma_n \end{array}$$

be the premises in the rule. For $i \in \{1, \ldots, n\}$ determine k_i, I_l, I_g and I_e such that $\{I_l, I_g, I_e\}$ is a partition of $\{1, \ldots, n\}$ and the following relations between the different γ_i and τ_i all hold.

$$\begin{array}{ll} \tau_i \equiv \iota_{i,k_i} \to \cdots \to \iota_{i,1} \to \gamma_i & i \in I_l \\ \gamma_i \equiv \iota_{k_i} \to \cdots \to \iota_1 \to \tau_i & i \in I_g \\ \tau_i \equiv \gamma_i & i \in I_e \end{array}$$

Then define

$$\begin{array}{rll} k_{\max} &= \max_{i \in I_g} k_i & I_g \neq \emptyset \\ k_{\max} &= 0 & I_g = \emptyset \\ y_i &= \pi_i e'_a & i \in I_e \\ y_i &= \lambda x_{i,k_i} \ldots \lambda x_{i,1}.\pi_i e'_a & i \in I_l \\ y_i &= (\pi_i e'_a)\ x_{k_i} \cdots x_1 & i \in I_g \end{array}$$

where all x_k and $x_{i,k}$ are fresh variables. More specifically, they belong to the sequence $\bar{x}$ and do not belong to $\mathrm{Dom}(A_{\bar{x}})$. The schema for the transformation rules is now given by

$$\frac{\begin{array}{c} A_{\bar{x}} \vdash e \rightsquigarrow e' : \tau_1 \times \cdots \times \tau_n \to \tau_r \\ A_{\bar{x}} \vdash e_a \rightsquigarrow e'_a : \gamma_1 \times \cdots \times \gamma_n \end{array}}{\begin{array}{c} A \vdash ee_a \rightsquigarrow \lambda x_{k_{\max}} \ldots \lambda x_1.e'(y_1, \ldots, y_n) : \\ \iota_{k_{\max}} \to \cdots \to \iota_1 \to \tau_r, \text{ when } k_{\max} > 0 \\ \tau_r, \text{ otherwise} \end{array}} [\mathit{TRANS}]$$

Fig. 5. Schema for transformation rules

to incorporate these kinds of overloading into implicitly typed languages with type inference.

The soundness of our inference systems with respect to the original Hindley-Milner systems is a pleasant property, since it means that the overloading introduced by our systems always can be resolved statically. This is is beneficial for implementations.

It is straightforward to adapt our inference systems to deal with overloading over data structures such as arrays or lists rather than functions. Only the transformational parts of the systems, given in Fig. 2 and Fig. 5, respectively, need to be changed as follows. Rather than matching function types for arguments, one must match the corresponding array or list types (i.e., in Haskell syntax, `Array a b` or `[b]` rather than `a -> b`). Furthermore, the rewritten expression must be formed with array- or list-forming primitives rather than through λ-abstraction. The way this is done will specify the semantics of the overloading. For instance, if the overloading is over arrays, then dynamic checks for the usual constraint that array arguments to elementwise overloaded operations must be *conformable* (have same extents in the corresponding dimensions) can be added.

The generality of the overloading given by our inference systems can be restricted in various ways without sacrificing our results. For instance, it is very straightforward to modify the transformational part of the systems so the elemental function overloading is performed only for a restricted set of operations. More interestingly, one can remove either the promotion, or the elemental function overloading, and the resulting systems will still be transparent and sound.

9 Future Work

The immediate next step is of course to develop an inference algorithm for the type inference and rewrite system for Core-ML. This would enable the practical use of elemental function overloading in implicitly typed languages with type inference. Such an algorithm should produce a "best" rewriting into a properly typed term whenever such a term exists. In order to do this, one must come up with some extended concept of principal type which relates also otherwise incomparable types which could result from a rewriting. Intuitively, this is easy: the "best" rewriting should produce an outermost function abstraction with a minimal number of variables. But the details remain to be worked out. It is also unclear for the moment what the practical complexity of such an algorithm will be. We believe that this complexity will be lower if promotion is left out, but again it remains to be shown that this is indeed the case.

Another topic of interest is how the kind of overloading produced by our systems can be made to coexist with other kinds of more conventional overloading, such as the class system in Haskell. A positive result would enable the introduction of elemental function overloading in Haskell. We think this would turn this language into a very convenient language for high-level specification and rapid prototyping of parallel numerical algorithms, and also open the door for the use of this kind of overloading in symbolic computing.

10 Acknowledgement

We would like to thank Karl-Filip Faxén for helpful discussions on various aspects of type systems and type inference in general and on our ideas in particular.

References

[1] Walter S. Brainerd, Charles H. Goldberg, and Jeanne C. Adams. *Programmer's Guide to FORTRAN 90*. Programming Languages. McGraw-Hill, 1990.

[2] S. M. Brien, M. Engel, He Jifeng, A. P. Ravn, and H. Rischel. Z description of duration calculus. Technical Report Draft, Oxford University Computing Laboratory, Programming Research Group, August 1993.

[3] Luca Cardelli. Basic polymorphic typechecking. *The ML/LCF/Hope News Letter*, II(1), January 1985. Also in *Science of Computer Programming*, 8:147–172, 1987.

[4] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.

[5] Keith Duddy, Luke Everett, Colin Millerchip, Brendan Mahony, and Ian Hayes. Z-based Notation for the Specification of Timing Properties. Technical report, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, June 1998. Draft, not published.

[6] Keith Duddy and Brendan Mahoney. *A lifting algorithm for interval history expressions*, 1994. Working paper.

[7] C. J. Fidge, I. J. Hayes, A. P. Martin, and A. K. Wabenhorst. A Set-Theoretic Model for Real-Time Specification and Reasoning. In Johan Jeuring, editor, *Proc. 4th Int. Conf. on Mathematics of Program Construction*, number 1422 in Lecture Notes in Computer Science, pages 188–206, Marstrand, Sweden, June 1998. Springer Verlag.

[8] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, March 1996.

[9] Per Hammarlund and Björn Lisper. On the relation between functional and data parallel programming languages. In *Proc. Sixth Conference on Functional Programming Languages and Computer Architecture*, pages 210–222. ACM Press, June 1993.

[10] Robert Harper and John C. Mitchell. On The Type Structure of Standard ML. Technical report, Carnegie Mellon University, Pittsburgh, PA 15213, January 1992.

[11] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1):1–170, June 1993. HPF Version 1.0.

[12] R. Hindley. The principal type-scheme for an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[13] K. E. Iverson. *A Programming Language*. Wiley, London, 1962.

[14] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, Cambridge, MA, 1994.

[15] Björn Lisper. Data parallelism and functional programming. In Guy-Reneé Perrin and Alain Darte, editors, *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, Vol. 1132 of *Lecture Notes in Comput. Sci.*, pages 220–251, Les Ménuires, France, March 1996. Springer-Verlag.

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

[17] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Proc. Functional Programming and Computer Architecture*, San Diego, California, June 1995. ACM Press.

[18] Eva Part-Enander, Anders Sjöberg, Bo Melin, and Pernilla Isaksson. *The MATLAB Handbook*. Addison-Wesley, 1996.

[19] Jay M. Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proc. IEEE*, 79(4):504–523, April 1991.

[20] Satish Thatte. A type system for implicit scaling. *Science of Computer Programming*, 17:217–245, 1991.

[21] Thinking Machines Corporation, Cambridge, MA. *Programming in *Lisp*, September 1988.

[22] Claes Thornberg. *Towards Polymorphic Type Inference with Elemental Function Overloading*. Licentiate thesis, Dept. of Teleinformatics, KTH, Stockholm, May 1999. Research Report TRITA-IT R 99:03.