Type-dependent Parameter Inference

G. V. Cormack & A. K. Wright†*
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

ABSTRACT

An algorithm is presented to infer the type and operation parameters of polymorphic functions. Operation parameters are named and typed at the function definition, but are selected from the set of overloaded definitions available wherever the function is used. These parameters are always implicit, implying that the complexity of using a function does not increase with the generality of its type.

1. Introduction

Function and data abstractions can be made more general by parameterizing them by *type*, rendering them polymorphic (cf. Cardelli and Wegner 1985). All but the most general abstractions must also be parameterized over certain specific operations whose semantics depend on the values of the type parameters. Although the type and operation parameters are necessary to the implementor of an abstraction, to the user they are largely redundant or irrelevant, and obscure the meaning of the abstraction. They provide a significant disincentive to the use of polymorphism: the more general a function is, the more burden the user incurs in using the function. It is desirable to supply these types and operations implicitly on the user's behalf. We argue that parameter inference is a good mechanism to achieve this end: type parameters are inferred in order to match the type required by context, and operation parameters are bound from an overloaded name space according to the type system.

Two well known languages that address different aspects of this problem are ML (Milner 1978, 1985) and Ada (1983). In ML, neither the definer nor the user of a function specifies type parameters; these are inferred automatically. No operation selection is done in conjunction with type parameter inference. In Ada, type parameters must be specified by the user of the function in a separate *instantiation* statement. Operation parameter selection and overload resolution are done automatically. In summary, ML infers type parameters but not operations; Ada infers operations but not type parameters.

Other languages, based on the polymorphic λ-calculus (Girard 1972, Reynolds 1974), have a more powerful type system than ML or Ada, and do some form of type inference. But the type inference rules are heuristic and do not apply uniformly in all contexts. Examples of such languages are Russell (Donahue and Demers 1985; Boehm 1986), Poly (Matthews 1985), IFX (O'toole and Gifford 1989) and ForceOne (Wright 1987; Cormack and Wright 1987).

Many language designs approach the problem of operation parameters by introducing another kind of object into the system, which packages operations together with the types on which they depend. Examples include classes in object-oriented languages (Meyer 1988), types in Russell, classes in Haskell (Wadler and Blott 1989), and higher order modules (functors in SML [MacQueen 1984], generic packages in Ada). These facilities are more complex and less flexible than type-dependent parameter inference.

2. Design Criteria

Although the language and examples presented here are small, we are interested in the programming-in-the-large aspects of polymorphism. In particular, we are interested in modularization, abstraction, and reusability. In the programming-in-the-large environment, abstractions and modules must be viewed from two perspectives: that of the implementor of the abstraction, and that of the user of the abstraction. In general, the user of an abstraction does not have the ability to modify the abstraction. Creating a copy of the abstraction to be modified is not a satisfactory method of reuse because it would create two variants to be maintained by separate authorities; for example, bug fixes to the original abstraction will not be propagated to the copy. Once an abstraction is in use, it may not be changed by the implementor in such a way as to invalidate potential existing uses. The implementor would have no authority to modify these applications to conform; neither is the implementor at liberty to inspect all users's programs to determine what use has been made of an abstraction.

We advance the following criteria as benefiting the cause of programming-in-the large:

Separate authority. We assume that each module is controlled by a separate authority, and the authority for one module is not able to modify another. When an abstraction is reused, we assume it is defined in a module separate from its use and must therefore be used without alteration.

* Electronic mail address: gvcormack@waterloo.edu
† Present address: Computer Science Department, Rice University, Houston, Texas 77251-1892. Electronic mail address: wright@rice.edu

Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation. White Plains, New York, June 20-22, 1990.

Generality. It should be possible to define an abstraction to have as large a domain of applicability as possible, in order to maximize the incidence of reuse of the abstraction. It should be possible to combine abstractions in an orthogonal manner to form new abstractions. This criterion is a primary motivation for polymorphism.

Generalizability. It should be possible for the implementor of an abstraction to increase its generality. So long as the modified abstraction is a generalization of the original, existing uses of the abstraction will not require change. It might be possible to modify an abstraction in a manner which is not a generalization without affecting existing uses, but, without inspecting the modules in which the uses occur, this possibility cannot be determined. This criterion precludes the addition of parameters, unless these parameters have default or inferred values that are valid for all possible existing applications.

Incrementality. The user of an abstraction should be able to augment its domain of applicability without modifying it. The use of subclasses in object-oriented languages is an example of incrementality.

3. Informal introduction and examples

Throughout this exposition, we shall use two central examples: the simple functional abstraction square that multiplies a value by itself, which introduces our type system and design criteria; and the more complex pair of related abstract type constructors matrix and vector, which illustrate a structural advantage of this system over parameterized modules.

Notation

We use a programming language notation that has constants, declarations, expressions, functions, user-defined types, and an information hiding facility. We assume a conventional set of constants, builtin functions, and types are available.

Function applications are of the form f(x), with syntactic sugar for operators: X *op* Y means *op*(X,Y). All parameter lists are curried; that is, f(x,y,z) means f(x)(y)(z).

Declarations are of the form

identifier: *prototype* = *expression*

where *prototype* defines the name and type of any parameters, and the overall type of *identifier*. Declarations may be anonymous by omitting *identifier*:, and : *prototype* may be omitted if *identifier* is to be the same type as *expression*.

A prototype consists of zero or more formal parameter declarations and a result type. Types may be constants, or type expressions of the form *type*→*type* (a function type), ∀*id type* (a polymorphic type), or *id*(*type*) (an instance of a type constructor). Formal parameters have three forms: [*id*: *type*] (an explicit parameter), [?*id*: *type*] (an implicit parameter), or [all *id*] (a type parameter).

Type constructors are types that are parameterized by other types, and are declared thus:

id: [*id*: type] type = *type* .

Information hiding is provided by bracketing a set of declarations within module *identifier* and end *identifier*. Within this module all declarations are visible, but outside it only the declarations prefixed by export are visible. Exported types and type constructors are *opaque*: they can be referenced outside the module as if they were built in; their definition cannot be referenced.

A functional abstraction

The specification for the functional abstraction square is: square is a function with one parameter, x, of any type such that x * x is defined, whose result is equal to x * x.

In our notation, a definition of square that meets the specification only for real values is:

```
square: [x: real] real = x * x
```

square has type

```
real→real
```

A valid application of square might be

```
square(1.23)
```

A polymorphic square that meets the specification is:

```
square: [all t, ?*: t→t→t, x: t] t = x * x
```

Here, t is a *type* parameter – whenever square is applied, it is bound to the type of the actual parameter for x. Throughout square, t may be used to refer to this type. ?* is an *implicit parameter* – it is bound automatically to an actual parameter of the same name. This binding is statically determined, however it *use-site biding*, rather than the usual *definition-site binding* of Algol-like languages.† The lexical scope of the application site is searched to find a definition of an identifier of the same name. All identifiers, including implicit parameters, are bound from an overloaded name space so as to match the type required by context. Possible valid applications of square are

```
square(pi)   -- yields real
square(2)    -- yields int
```

In the first example, t is bound to real, and then the implicit parameter ?*: real→real→real is resolved to the built-in real multiplication operator. In the second application, t is bound to int and integer multiplication is used. We use the following annotations to illustrate the inferred information: inferred parameters are enclosed in braces {}, and overloaded identifiers are identified by a unique subscript.

```
square{real,*real}(pi)
square{int,*int}(2)
```

The above applications illustrate the criterion of generalizability: the generalized square is valid (without modification) in all contexts where the more specific square was valid.

The following application illustrates the criterion of incrementality:

```
(
  *: [a: string, b: string] string = concat(a,b);
  square("abc")  -- yields "abcabc"
)
square("abc")    -- illegal; appropriate * not visible
```

The function square may be applied to string values (as in formal language theory) without modifying either the module defining square, or the module defining the type string, even though the definition of string does not include an operator *. Within a local scope, it is possible to define such a suitable multiplication operation, and to have the more globally defined square apply within that

† Further discussion of this binding mechanism is presented elsewhere (Cormack 1983, Cormack and Wright 1987).

scope. Outside the scope, square cannot be applied to strings. In effect, the local definition of * augments all abstractions that depend on *.

In other application domains, the multiplication operator is used to specify composition of functions. That is, (f*g)(x) means g(f(x)). It is possible to specify such a definition:

```
*func: [all a, all b, all c, f:a→b, g:b→c] a→c =
    [x:a] c = g(f(x))
```

In the scope of this definition, square can be applied to functions, and the composition operator will be applied: square(log) is annotated as

```
square{real→real,*func{real,real,real}}(log),
```

and the resulting type is real→real (a function that applies log twice to its argument). First, x (the parameter to square) is bound to log, and t is bound to the type of log, i.e. real→real. Second, the implicit parameter * with type (real→real)→(real→real)→real→real is selected from the environment. The above definition $*_{func}$ does not match exactly – it must be *specialized* by binding a, b, and c all to real, thus yielding the desired type. In general, actual parameters may be specialized by applying type parameters or implicit parameters in order to match the desired formal type. It is possible to apply square to a function with implicit parameters; for example to square itself:

```
square(square)
```

Here, each instance of square requires an implicit parameter *, but the two implicit parameters have different types. From the set of available definitions (that is, $\{*_{func}, *_{real}, *_{int}\}$), only $*_{func}$ has an appropriate type for the first square, but any of the three is possible for the second. If the result type is constrained by context, the appropriate version will be selected:

```
quad: int→int =
      square{int→int,*func{int,int,int}}
        (square{int,*int})
quad: real→real =
      square{real→real,*func{real,real,real}}
        (square{real,*real})
quad: [all a] (a→a)→a→a =
      square{(a→a)→a→a,*func{a→a,a→a,a→a}}
        (square{a→a,*func{a,a,a}})
quad: [all t, ?*t: t→t→t] t→t =
      square{t→t,*func{t,t,t}} (square{t,*t})
```

The last definition is the most general; it has the same domain of applicability as square itself. Finally, we observe that square, as defined above, is not as general as possible. Instead, square could have been defined thus:

```
square: [all a, all b, ?*: a→a→b, x: a]b = x * x
```

As dictated by the criterion of generalizability, all of the above examples apply without modification if this change is made, as well as a number of new applications. For example,

```
*list: [all a, x: a, y: a] list(a) = ...
a: list(int) = square(3)                    -- uses *list
b: int = square(3)                          -- uses *int
x: list(list(int)) = square(square)(3)      -- uses *list
y: int = square(square)(3)                  -- uses *int
```

A data abstraction

We illustrate the applicability of our type system for data abstraction by defining two polymorphic abstract data types, vector and matrix, specified as follows:

1. vector is a type constructor with one parameter, the element type in the vector; similarly, matrix is a type constructor with one parameter specifying its element type.
2. For simplicity, the size of all vectors and matrices is the same, and denoted by the global constant size.
3. The dot-product of any two vectors A and B can be taken using the operator *, provided that

   a * b + a' * b' + ...

   is valid for a an element of A, and b an element of B; similarly, any pair of matrices or any pair consisting of one matrix and one vector are multiplied by * providing the same constraint applies to their elements.
4. The sum of any two vectors or any two matrices is taken by the operator +, provided the elements are also summable using +.
5. These are the only operations applicable to matrices and vectors; in particular, neither access to the implementation nor element-by-element access is to be allowed outside their defining module.

Figure 1 presents a realization of these abstractions, encapsulated in a module with exported definitions for the type constructors vector and matrix, the four kinds of multiplication among vectors and matrices, and the two kinds of addition. Our implementation uses functions to represent vectors and matrices, but the user is unaware of this implementation, because exported types are opaque. The functions row and col are internal functions not visible outside the module. Using type parameters, the functions row, col, *, and + are defined to apply among the various combinations of vectors and matrices. Figure 2a shows a number of applications of linear_algebra, and figure 2b shows the annotation for selected examples. The first six applications illustrate in a straightforward way how the functions * and + apply, and extract the element types using type parameters.

The remaining applications illustrate capabilities that are absent from other languages. mv1 * mv2, involves a recursive vector multiplication: the outer application multiplies values of type vector(vector(real)), while the recursive application involves vector(real). Also, the result of mv1 * mv2 is a matrix with a different element type from either mv1 or mv2. We call * a *transcendental* function because its parameters and result can be of more than one instance of a type constructor. The application mm * mv2 illustrates again the ability of a function to transcend instances of a type constructor. In this case, the two parameters have different instances of the matrix type constructor; namely, matrix(matrix(real)) and matrix(vector(real)). The application, x * i provides a simpler example of a transcendental invocation of *. In this case, a vector(real) is multiplied by a vector(int) provided the system (or user) has defined multiplication of a real by an int. The final two applications are of the previously defined square to a matrix: the first is type consistent; the second isn't.

Figure 1. Polymorphic linear algebra package.

```
module linear_algebra
  export vector: [t: type] type = (1..size)→t;
  export matrix: [t: type] type = (1..size)→(1..size)→t;

  row: [all t, m: matrix(t), i: 1..size] vector(t) =
        [j: 1..size] t = m(i,j)
  col: [all t, m: matrix(t), j: 1..size] vector(t) =
        [i: 1..size] t = m(i,j)

  export *1: [ all at, all bt, all rt,
               ?*: at→bt→rt, ?+: rt→rt→rt,
               a: vector(at), b: vector(bt)
             ] rt = ( s: var rt := a(1) * b(1);
                      for i:int from 2 to size do
                        s := s + a(i) * b(i);
                      endfor;
                      s
                    )
  export *2: [ all at, all bt, all rt,
               ?*: at→at→bt, ?+: rt→rt→rt,
               a: matrix(at), b: matrix(bt)
             ] matrix(rt) =
             [i: 1..size, j: 1..size] rt = row(a,i) * col(b,j)
  export *3: [ all mt, all vt, all rt,
               ?*: mt→vt→rt, ?+: rt→rt→rt,
               m: matrix(mt), v: vector(vt)
             ] vector(rt) =
             [i: 1..size] rt = row(m,i) * v
  export *4: [ all vt, all mt, all rt,
               ?*: vt→mt→rt, ?+: rt→rt→rt,
               v: vector(vt), m: matrix(mt)
             ] vector(rt) =
             [j: 1..size] rt = v * col(m,j)
  export +1: [ all at, all bt, all rt,
               ?+: at→bt→rt,
               a: vector(at), b: vector(bt)
             ] vector(rt) =
             [i: 1..size] rt = a(i) + b(i)
  export +2: [ all at, all bt, all rt,
               ?+: at→bt→rt,
               a: matrix(at), b: matrix(bt),
             ] matrix(rt) =
             [i: 1..size, j: 1..size] rt = a(i,j) + b(i,j)
end linear_algebra
```

Figure 2a. Applications of linear algebra.

```
a,b,c: var matrix(real);
x,y,z: var vector(real);
r:     var real;

r := x * y;   -- vector(real) * vector(real) ⇒ real
c := a * b;   -- matrix(real) * matrix(real) ⇒ matrix(real)
z := a * y;   -- matrix(real) * vector(real) ⇒ vector(real)
z := x * b;   -- vector(real) * matrix(real) ⇒ vector(real)
z := x + y;    -- vector(real) + vector(real) ⇒ vector(real)
c := a + b;    -- matrix(real) + matrix(real) ⇒ matrix(real)

mv1, mv2: var matrix(vector(real));
mm: var matrix(matrix(real));
i: vector(int);

a := mv1 * mv2;
             -- matrix(vector(real)) * matrix(vector(real))
             -- ⇒ matrix(real)

mv1 := mm * mv2;
             -- matrix(matrix(real)) * matrix(vector(real))
             -- ⇒ matrix(vector(real))

r := x * i;      -- works if we can multiply  real * int

mm := square(mm);  -- square from example 1

mv1 := square(mv1);
             -- invalid; can't bind
             -- ?*: vector(real)→vector(real)→vector(real)
```

Figure 2b. Annotated applications.

```
r := *1{real,real,real,*real,+real}(x,y);
c := *2{real,real,real,*real,+real}(a,b);
z := *3{real,real,real,*real,+real}(a,y);
z := *4{real,real,real,*real,+real}(x,b);
z := +1{real,real,real,+real}(x,y);
c := +2{real,real,real,+real}(a,b);

a := *2{vector(real),vector(real),real,
        *1{real,real,real,*real,+real},
        +1{real,real,real,+real}
       }(mv1,mv2);

mv1 := *2{matrix(real),vector(real),vector(real),
         *3{real,real,real,*real,+real},
         +1{real,real,real,+real}
        }(mm,mv2);

mm := square{matrix(matrix(real)),
           *2{matrix(real),matrix(real),matrix(real),
             *2{real,real,real,*real,+real},
             +2{real,real,real,+real}
            }
          }(mm);
```

Discussion of language features

An obvious alternative to the mechanism proposed here is to abandon overloading and use full explicit parameterization. For example, an alternate version of square may be written and invoked as

```
square: [t: type, *: t→t→t, x: t] t = x * x
square(real, realmultiply, 2.0)
square(int, intmultiply, 2)
```

These extra parameters are perhaps a minor inconvenience here, but if parameterization is explicit, layered or higher order abstractions can be used only with great difficulty. Consider the analogue to the application square(square):

```
quad: [t: type, *: t→t→t, x: t] t =
  (
    squaret: [y:t]t = square(t,*,y);
    mulsquare: [f: t→t, g: t→t] = compose(t,t,t,f,g);
    square(t→t, mulsquare, squaret)(x)
  )
```

The existence of these extra parameters means that the greater the generality of an abstraction, the greater the burden of use on the programmer, and directly conflicts with our design criteria of generalizability and incrementality – square cannot be applied to new types unless a special interface function is written for each new type. ML, Ada, and Russell, because they infer only a subset of types and operations, do not meet our criteria.

Two recent articles have identified weaknesses in ML and proposed solutions that are similar to each other, and similar to the solution presented here. Kaes (1988) proposes that overloaded symbols be defined in advance, and that all symbols of the same name have a common polymorphic type. Instances of the symbol name are assigned values that are specializations of the common type. Only functions may be overloaded; the type of any instance of a name must be inferable from the type of its first parameter. In our notation, this means that implicit parameters can be inferred only *between* a function and its argument in an application (i.e. f{t}(x)), and the type required by context is not considered. This approach supports a measure of incrementality and generalizability, but the restrictions preclude the specification of most of the examples in this paper.

Wadler and Blott (1989) propose a similar mechanism, which has been incorporated in the language Haskell. Instead of identifying individual symbols to be overloaded, classes of symbols are specified. Nevertheless, a particular symbol may belong to only one class, and therefore the class mechanism provides a grouping facility but no additional expressive power. Wadler and Blott's inference rules appear to be more powerful than those of Kaes, but no algorithm is given to handle overloaded constants, or to perform overload selection that depends on more than one parameter type. Overloaded names must be pervasive: they cannot be hidden using normal scope rules. The authors further propose that all overload declarations be required to have global scope. We are unable to reconcile this requirement for pervasive declarations with our criteria for separate authority and incrementality.

A large number of languages rely on selecting operations from the module defining an abstract data type. To illustrate this approach, we give an alternate definition of square in an imaginary dialect of our language:†

```
square: [all t containing(*: t→t→t), x: t] = x * x
```

In this example, we use the type parameter t as before, with the additional clause specifying that the module defining t must contain a suitable definition for *. Presumably, the built-in types real and int can be considered to contain *, so the following applications are correct:

```
square(1)
square(2.0)
```

On the other hand, the module defining string does not supply an operator named *, so it is impossible to apply

```
square("abc")
```

no matter what local definitions are provided – the only way to apply square to strings is to modify the module defining string. Therefore the criteria of incrementality and separate authority cannot both be met.

Higher order abstractions illustrate further problems with using abstract data types to supply operations. Defining * to be composition for functions presents no particular problem. However, applying square to a function does:

```
square(log)
```

Here, t is bound to the type real→real, which must contain a definition of *. In this case, there is no identifiable module defining real→real; it is just an instance of a built-in type constructor. There is no mechanism to augment such a type.

Figure 3. A parameterized module.

```
module linear_algebra [t:type containing(*:t→t→t, +:t→t→t)]
  export vector: type = (1..size)→t;
  export matrix: type = (1..size)→(1..size)→t;

  row: [matrix, i: 1..size] vector =
        ([j: 1..size] t = m(i,j))
  col: [all e, m: matrix, j: 1..size] vector =
        ([i: 1..size] t = m(i,j))

  export *1: [a: vector, b: vector] t = (
                s: var rt := a(1) * b(1);
                for i:int from 2 to size do
                  s := s + a(i) * b(i);
                endfor;
                s
              )
  export *2: [a: matrix, b: matrix] matrix =
          ([i: 1..size, j: 1..size] t = row(a,i) * col(b,j))
  export *3: [ m: matrix, v: vector ] vector =
          ([i: 1..size] t = row(m,i) * v)
  export *4: [v: vector, m: matrix] vector =
          ([j: 1..size] t = v * col(m,j))
  export +1: [a: vector, b: vector] vector =
          ([i: 1..size] t = a(i) + b(i))
  export +2: [a: matrix, b: matrix] matrix =
          ([i: 1..size, j: 1..size] t = a(i,j) + b(i,j))
end linear_algebra
```

Parameterized modules or classes have been used as the basis for polymorphism in many languages; for example, Ada (1983), OBJ2 (Futatsugi et al. 1985), SML (MacQueen 1984), Eiffel (Meyer 1988), and Quest (Cardelli 1989). Grouping types and operations into modules provides a shorthand for passing them to other abstractions; nevertheless, they must be instantiated and the instances passed explicitly to other abstractions. Parameterized modules cannot be used to define private transcendental functions such as those defined in linear_algebra in figure 1. A straightforward attempt to define linear_algebra as a parameterized module appears in figure 3. Instead of parameterizing the type constructors vector and matrix, we parameterize the module. vector and matrix become types rather than type constructors, and we replace all occurrences of the form vector(α) or

† We shall denote deviations from our language using italics.

matrix(α) by vector and matrix respectively. This module is simpler than that of figure 1, but it is also much more restrictive. The types of all of the operations have been restricted so that they apply to and return only matrices and vectors with a common element type. These definitions are not general enough to satisfy the examples in figure 2b, and they cannot be generalized to do so. In general, it is impossible to define within a parameterized module a function that applies to other instances of the same module. The only possibility is to define such functions outside (which would involve exporting row, col, and further private operations). This defeats information hiding, and also defeats the module-oriented method of operation selection (i.e. *containing(...)*).

In languages based on the polymorphic λ-calculus, it is possible to associate run-time information with data types. It is a simple matter to package a number of operations with the representation of a type; this approach is used in Russell. It imposes the structural constraint that data types must *contain* their operations. The benefit of this approach is that it is possible to have dynamic typing which our system cannot accomplish. The disadvantages are essentially the same as noted above for module-based polymorphism: the type cannot be augmented with local operations, and private polymorphic operations can refer only to individual instances of parameterized types.

4. A more formal description

We define here two tiny languages, ϵ and χ, which represent the essential capabilities of the type system. ϵ has overloading and implicit parameter and type application. χ has explicit overload resolution, and explicit type and parameter application. Type rules are given for χ, and ϵ is defined in terms of replacement rules that transform ϵ expressions into χ expressions.

ϵ consists of expressions containing identifiers, function abstractions, and function applications. An abstract syntax for expressions in ϵ is given in figure 4.

Figure 4. Abstract syntax for ϵ.

```
e ::= t | id = e; e | id: def; e
t ::= id | t t | def
def ::= type = e
      | [id: type] def
      | [all type_id] def
      | [?id: type] def
type ::= type_id | type → type | ∀ type_id type
```

In χ, every identifier is subscripted by a unique qualifier. Type applications are made explicit, as are implicit parameter applications. Type applications are restricted to exclude first-level quantified types (thus avoiding ambiguity in the inference process).† An abstract syntax for χ is given in figure 5. An expression in ϵ is transformed into an equivalent expression in χ by repeated application of the rules in figure 6, subject to the signature and type rules stated in the remainder of this section.

Modules, and user-defined types and type constructors, have been excluded; modules provide visibility control only, and user-defined type constructors are handled in the same manner as the builtin constructor →. Function abstractions and applications are represented using their curried form.

† Without this restriction, in the expression
ident: [all a][x:a] a = x; ident ident,
ident ident could be transformed either to
ident (∀ a→a) ident, or to
[all a] a→a = ident (a→a) (ident (a))

Figure 5. Abstract syntax for χ.

```
e ::= t
    | id_qual = e ; e
    | id_qual: def ; e
t ::= id_qual
    | t t
    | t {simple_type}
    | t {t}
    | def
def ::= type = e
      | [id_qual: type] def
      | [all type_id] def
      | [?id_qual: type] def
type ::= type_id | type → type | ∀ type_id type
simple_type ::= type_id | type → type
```

Figure 6. Transformation rules from ϵ to χ.

Overload resolution:	id ⇒	id_{qual}
Type application:	t ⇒	t { simple_type }
Implicit application:	t_1 ⇒	t_1 { t_2 }
Type qualification:	t ⇒	type = t
Type abstraction:	def ⇒	[all type_id] def *(type_id not free in def)*
Implicit abstraction:	def ⇒	[?id_{qual}:type] def *(id_{qual} not free in def)*

Signature checking in χ

Each term t in χ has a *signature* and a *name*. The signature of a term represents both its type and its implicit parameterization. A signature has the following abstract syntax:

```
sig ::= type
      | type → sig
      | ?id:type → sig
      | ∀ type_id sig
```

The name of a term is an identifier that is used to constrain the binding of implicit parameters. Type and implicit parameter applications and abstractions preserve the name of a term. Function applications and abstractions yield the name anon, which is not equal to any identifier. The function *nameof* computes the name of any term:

```
nameof(id_qual) = id
nameof(t { simple_type }) = nameof(t)
nameof(t1 { t2 }) = nameof(t1)
nameof(t1 t2) = anon
nameof(type = e) =
  if e = t then nameof(t) else anon
nameof([id_qual: type] def) = anon
nameof([?id_qual: type] def) = nameof(def)
nameof([all type_id] def) = nameof(def)
```

In χ, a signature environment maps identifiers to their signatures:

```
env ::= empty | env <id_qual, sig>
```

The function *lookup* returns the signature of an identifier or invalid:

lookup(env <$id1_{qual1}$, sig>, $id2_{qual2}$) =
  if $id1_{qual1}$ = $id2_{qual2}$ then sig else lookup(env, $id2_{qual2}$)
lookup(empty, id_{qual}) = Invalid

Predefined constant values in χ are defined in the signature environment standard: for every constant c with type t, lookup(standard, c) = t.

The signature of a term with respect to a signature environment is yielded by the function *sigof*. Identifiers are looked up in the environment. The parameter and argument types for function applications must match (using the comparison $=_\alpha$, which permits systematic renaming of non-free type identifiers) and the function's result signature is yielded. Type application involves substitution of a specific type in place of the type parameter to the function (the notation *[x:=y]* indicates a uniform replacement of *x* by *y*, with renaming to avoid capture of type identifiers). The application of implicit parameters involves matching both the type and name of the formal and actual parameters, and yielding the result signature. The signatures of function, type, and implicit abstractions are built from their prototype definitions by the function *sigdef*.

sigof(env, id_{qual}) = lookup(env, id_{qual})
sigof(env, t1 t2) =
  if sigof(env, t1) = type → sig and
  sigof(env, t2) $=_\alpha$ type
  then sig else Invalid
sigof(env, t1 {simple_type}) =
  if sigof(env, t1) = ∀type_id sig
  then sig *[*type_id:=simple_type*]*
  else Invalid
sigof(env, t1 {t2}) =
  if sigof(env, t1) = ?id:type → sig
  and nameof(t2) = id
  and sigof(env, t2) $=_\alpha$ type
  then sig else Invalid
sigof(env, def) = sigdef(env, def)

sigdef(env, type = e) = type
sigdef(env, [id_{qual}: type] def) =
  type → sigdef(env <id_{qual}, type>, def)
sigdef(env, [all type_id1] def) =
  ∀type_id2 sigdef(env, def *[*type_id1:=type_id2*]*)
    *(*type_id2 *not free in* env*)*
sigdef(env, [?id_{qual}: type] def) =
  ?id:type → sigdef(env <id_{qual}, type> , def)

Expressions have types rather than signatures; that is, they may not have implicit parameters. Expressions can appear in either completely constrained or completely unconstrained contexts. In a completely constrained context (type = e), the type of an expression must equal the constraint type. In an unconstrained context, the expression can yield any type. The function typeof computes the type of an expression, using the auxiliary function checktype to verify that expressions in constrained contexts yield the appropriate type.

typeof(env, t) =
  if sigof(env, t) = type then type
  else Invalid
typeof(env, id_{qual} = e1; e2) =
  typeof(env <id_{qual}, typeof(env, e1)>, e2)
typeof(env, id_{qual} : def; e) =
  if checktype(env <id_{qual}, sigdef(env, def)>, def)
  then typeof(env <id_{qual}, sigdef(env, def)>, e)
  else Invalid

checktype(env, type = e) =
  typeof(env, e) $=_\alpha$ type
checktype(env, [id_{qual}:type] def) =
  checktype(env <id_{qual}, type>, def)
checktype(env, [all type_id1] def) =
  checktype(env, def *[*type_id1:=type_id2*]*)
    *(*type_id2 *not free in* env*)*
checktype(env, [?id_{qual}: type] def) =
  checktype(env <id_{qual}, type>, def)

Overall, a χ expression e is type correct if typeof(standard, e) ≠ Invalid.

Signature checking in ϵ

An expression in the language ϵ is well typed if and only if it can be transformed into a well typed χ program by repeated application of the rules in figure 6. These transformation rules may not yield a unique χ expression from a given ϵ expression: several different overload resolutions may be possible, or, for a given overload resolution, several typings may be possible. Often, the variants are only trivially different, or there is a well defined *best* solution. If the variants differ nontrivially and there is no best solution, the ϵ expression is considered ambiguous. The rules for selecting the best solution (to be formalized in the next section) are:

Smallest solution. Within an expression, overload resolution is done so as to minimize the overall number of implicit parameter applications.

Most specific overload resolution. In a given context, if a polymorphic function and a more specific function (see next section) with the same name are both type consistent, the more specific function is selected.

Most general intermediate type. Given a unique overload resolution, the most general intermediate type is chosen for each term.

Generality of types and signatures

A type t1 is more general than t2 if a term having type t1 can replace a term having type t2 in any context. That is, t1 is a subtype of t2. In χ, a polymorphic type is never more general than a monomorphic type, because additional type parameters must be specified. In ϵ, the use of implicit parameters establishes a generality relationship among polymorphic and monomorphic types. For example, the type ∀a a→a is more general than the type Int→Int, because any term t of type ∀a a→a can be transformed automatically to the type Int→Int by the type application t{Int}. Similarly, the type ∀b∀c b→c is more general than ∀a a→a because a term t of the first type can be transformed using type abstraction and two type applications:

[all c] c→c = t{c}{c}.

In general, type application specializes the type of a value.

Type abstraction as specified in figure 6 neither generalizes nor specializes, but is an equivalence. That is, for any term t of type q (in which q does not contain r), the term [all r]q = t is applicable in exactly the same contexts as t.

Within a particular signature environment, implicit parameter application and abstraction are both equivalences, provided the environment contains an appropriate definition for the implicit parameter. A term t with signature ?x:a→b can be converted to the signature b by the implicit parameter application t{x_{qual}} if and only if <x_{qual}, c> is in the environment, and c is more general than a. Similarly, a term t with signature b can be converted to a term with signature ?x:a→b by the abstraction [?x_{qual}:a]b = t.

To the programmer, these observations have the following effect on generalizability. First, functions and values can be generalized with respect to type, and remain generally applicable. Second, functions and values can be given implicit parameters, provided a default value with the appropriate name is entered into the same environment. For example, the function

square: [i:int]int = ...

can be replaced by the two definitions:

square: [all t, ?times: t→t→t, i:t]t = ...
times: [x:int, y:int]int = ...

Any environment that contains square will also contain this definition of times, and therefore the new definition is applicable everywhere the previous definition is applicable.

Parameter inference rules

We first consider parameter inference independent of overload resolution; that is, we assume an oracle supplies the correct overload selection for any identifier. We then convert this nondeterministic algorithm to a deterministic one using an overload resolution algorithm adapted from Ada (cf. Aho, Sethi and Ullman 1986).

Parameter inference for terms. The first step in parameter inference is to compute the most general signature for each term in the expression. A term is either an identifier, a definition, or an application. The signature of an identifier is simply determined from the environment; the signature of a definition is given by the function *sigdef* defined previously. The signature of an application must be inferred.

Consider the application f x where f has signature fs and x has signature xs. fs must have the general form

$$\forall ft_1 \ldots \forall ft_i\ ?fq_1{:}fr_1{\rightarrow} \ldots\ ?fq_j{:}fr_j{\rightarrow}\ (\forall at_1 \ldots \forall at_k\ \alpha) \rightarrow \beta$$

That is, f is a function with $i \geq 0$ type parameters, $j \geq 0$ implicit parameters,† whose parameter type is α with $k \geq 0$ type parameters, and whose result signature is β. As a shorthand we use $X_{i=1}^{n}$ to denote X repeated for i from 1 to n; using this shorthand, fs is:

$$(\forall ft_n)_{n=1}^{i}\,(?fq_n{:}fr_n{\rightarrow})_{n=1}^{j}\,((\forall at_n)_{n=1}^{k}\,\alpha){\rightarrow}\beta$$

x has the type γ with $l \geq 0$ type parameters and $m \geq 0$ implicit parameters:

$$(\forall xt_n)_{n=1}^{l}\,(?xq_n{:}xr_n{\rightarrow})_{n=1}^{m}\,\gamma$$

Let *free* be the set of free type variables in fs and xs.

Signature inference for f x is based on the first-order unification (Robinson 1965; cf. Aho, Sethi and Ullman 1986) of α and γ, where the set of variables is $\{ft_n\} \cup \{xt_n\}$. We find the most general unifier *mgu* such that $mgu(\alpha) =_\alpha mgu(\gamma)$. If no *mgu* exists, there is no valid typing for f x, but the existence of *mgu* does not guarantee a valid typing. In particular, $mgu(ft_n)$ must be a simple_type expression whose free references are in *free*, and $mgu(xt_n)$ must be a simple_type expression whose free references are in $free \cup \{at_n\}$. If the bindings of *mgu* obey these rules, we transform f x to the following term:

$[\text{all } ft_n]_{n=1}^{i}$
$[\text{all } xt_n]_{n=1}^{l}$
$[?fq_n{:}mgu(fr_n)]_{n=1}^{j}$
$[?xq_n{:}(\forall at_p)_{p=1}^{k}\,mgu(xr_n)]_{n=1}^{m}$
$mgu(\beta) =$
f
$\{mgu(ft_n)\}_{n=1}^{i}$
$\{fq_n\}_{n=1}^{j}$
$[\text{all } at_n]_{n=1}^{k}$
$mgu(\alpha) =$
x
$\{mgu(xt_n)\}_{n=1}^{l}$
$\{xq_n\{at_p\}_{p=1}^{k}\}_{n=1}^{m}$

The type abstractions of the form [all t] def, where t is not free in def, are useless (but not incorrect), and can be eliminated.

It is straightforward to show that the above term has a valid signature. Some intuition can be gained by examining representative cases. Suppose fs is of the form $\forall t\ \alpha{\rightarrow}\beta$ and xs is γ. If $mgu(t) = t$ after unifying α and γ, the most general typing results from abstracting the entire term with respect to t:

[all t] $mgu(\beta)$ = f {t} ($mgu(\alpha)$ = x)

or, more simply:

[all t] $mgu(\beta)$ = f {t} x

If t is bound to some type q, the resulting term is:

[all t] $mgu(\beta)$ = f {q} x

which may be simplified to

f {q} x

Type parameters in xs are handled in an analogous manner, except for the following situation. Consider fs of the form $(\forall t\ \alpha){\rightarrow}\beta$. In this case, xs must be of the form $\forall q\ \gamma$. Unifying α and γ might yield $mgu(q) = t$, which would result in the term

f ([all t] $mgu(\alpha)$ = x {t})

If f and x had the signatures fs = ∀a(∀b a→b→ b)→a, and xs = ∀c∀d∀e c→d→e, the following term would result:

```
[all a][all c][all d][all e] a =
    f {a} ([all b] a→b→b = x {a}{b}{b})
```

† The type parameters need not precede the implicit parameters, but the transformation rules can be used to effect this ordering.

which could be simplified to

[all a] a → f {a} ([all b] a→b→b → x {a}{b}{b})

On the other hand, if xs = ∀c c→c→c, a would be bound to b (via c), violating our restriction. Without the restriction, the resulting term would be

[all a] b → f {b} [all b] x {b}

which has no valid signature because the first two occurrences of b are free references, while the last two are not. They could not possibly be equal, as required by the signature rules.

The treatment of implicit parameters is similar: because the environment of f x is identical to the environment of f, any implicit parameters to f can be moved outside f x by abstracting f x with respect to a parameter of identical name and type, and applying that parameter to f. Consider the signatures fs = ?q:t→a→b and xs = a; the resulting term is:

[?q:t]b → f{q}x

Implicit parameters to x are handled in a similar manner, except that the type environment of x, while initially identical to that of f x, can be augmented by the transformation process. Consider the case of fs = (∀a a→ a)→b and xs = ∀c ?q:(c→c) c→c. a→a and c→c unify with mgu(c) = a. Because the type a is known only in the environment of x, there cannot possibly exist a function q whose type is a→a. The most specific type that can match a→a is ∀a a→a. We therefore abstract f x with respect to an implicit parameter q of type ∀a a→a:

[?q:∀a a→a] b → [all a] x {a}{q{a}}

This transformed f x is applicable in exactly the same contexts as the original.

Parameter inference for expressions. When a term t is used as an expression, type parameters and implicit parameters may be applied. If the expression e appears in a context constraining it to be of type α (i.e. the expression α = e) we specialize t as necessary by (in effect) applying the identity function to t:

([x:α]α → x) t

Once the type is so specialized, it will have a signature of the form

$(\forall a_n)_{n=1}^{i}\ (?q_n{:}r_n\rightarrow)_{n=1}^{j}\ \alpha.$

An expression cannot have implicit parameters, so they all must be applied (no implicit parameters were removed by the rules for terms – they were merely promoted to form part of the result signature). The implicit parameters are applied one at a time until the resulting signature has no more implicit parameters. That is, so long as the term t has a signature of the form

$(\forall a_n)_{n=1}^{i}\ (?q_n{:}r_n\rightarrow)_{n=1}^{j\geq 0}\ \alpha$

we convert q_1 to an explicit parameter; that is, we (in effect) replace t by

$([\text{all } a_n]_{n=1}^{i}\ [q_1{:}r_1]\ [?q_n{:}r_n]_{n=2}^{j}\ \alpha \rightarrow$
$\quad t\{a_n\}_{n=1}^{i}\{q_1\}\{q_n\}_{n=2}^{j})\ q_1\ ,$

and apply the transformation algorithm recursively. Once the implicit parameters have been applied, the resulting term can be simplified by β substitution to remove many inferred type and implicit parameter abstractions.

The recursive application of implicit parameters may not terminate: when q_1 is bound in the signature environment of the expression, its signature may have implicit parameters; these parameters, when bound, may have more implicit parameters, and so on. Our implementation restricts the number of recursive applications to some constant k. Thus, the algorithm constructively determines whether, for a given ϵ expression of size s, there is an equivalent χ expression of size $s+k$ or smaller, where size is defined as the number of terms in the expression.

Overload resolution. The algorithm above assumes that the bindings for overloaded identifiers are known. In fact, we must label each definition of an identifier with a unique qualifier, and then add an appropriate qualifier to each identifier reference. To handle overload resolution, a two pass algorithm is used. The first pass computes the set of possible signatures for each term, and determines whether a valid type exists for each expression. Each recursive step to resolve implicit parameters takes a set of possible signatures, and applies one implicit parameter to each at any given recursive step. One of three termination conditions may occur: the set of possible signatures may become empty, in which case no valid χ program can be generated. After n ($0 \leq n \leq k$) steps of implicit parameter application, exactly one signature may have no implicit parameters. This is the result type of the smallest solution. If more than one signature has no implicit parameters, the expression is ambiguous (but we may choose to select, for example, the most general result type, if there is one). If after k steps, all possible signatures have at least one implicit parameter, there is no solution of size smaller than $s+k$.

Once a result type is chosen, the first pass is reversed, and the appropriate qualifiers are added to identifier references. Type and implicit parameter application and abstraction are inserted, transforming the expression to χ. (Alternatively, all typing can be erased to yield an equivalent expression in an untyped language; once overload selection is done, types are redundant.) It is possible to discover ambiguity during this second pass: several possible bindings may yield the desired type. In this event, we select the best fit, if it exists and is unique. We consider a term of the form $f_1\ x_1$ to be a better fit than a term of the form $f_2\ x_2$ if the signature of $f_1\ x_1$ is more specific or equivalent to the signature of of $f_2\ x_2$, and f_1 is a better fit than f_2 and x_1 is a better fit than x_2.

Discussion of the algorithm

The algorithm for type-dependent parameter inference incorporates three sub-algorithms that have not previously been combined: type inference, implicit parameter binding, and overload resolution.

Two major variants of type inference have been discussed in the literature: type inference for ML-like languages, and type inference for languages based on the polymorphic λ-calculus. ML-style inference expressed as a first-order unification problem. Our inference is more general than ML in that higher-order polymorphic functions may be written, but more restrictive in that function abstractions must be explicitly typed. Mitchell (1988) describes a number of variants of the polymorphic type inference problem. Our algorithm addresses a new subset of pure type inference without retyping functions; our restrictions are: function abstractions must be typed explicitly, and we restrict type application to apply only to simple types. The second restriction can be removed at the expense of creating ambiguous typings, but we have not

investigated removing the first. From a software engineering point of view, we have little incentive to do so, as we believe function interfaces should be specified independent of their implementations.

Others have proposed restricted polymorphic inference algorithms. Boehm (1986) describes such an algorithm for the language Russell, but states that it is difficult to define precisely the set of programs for which the algorithm will succeed. Boehm (1989) describes a more well defined set of rules that require the following restrictions: (1) type parameters are inferred only when functions are applied (and not when functions are passed unapplied to higher order functions); (2) functions must be uncurried so that type parameters are applied in conjunction with other parameters that depend on them. O'toole (1989) proposes a different approach, allowing an explicit conversion between ML-style polytypes and type abstractions. In contexts where a type abstraction is explicitly converted to an ML-style polytype, type inference is done. Elsewhere, type applications must be explicit.

Boehm (1985) describes partial polymorphic type inference, and shows it to be undecidable. In his characterization of inference, some but not all typings may be specified, and the remainder are inferred. Boehm's characterization disallows the inference of type abstractions, and requires that the positions of omitted type parameters be explicitly marked. The proof of undecidability relies on these restrictions.

Ada has implicit operation parameters to generic functions and packages. However, the type parameters to generics must be specified first, and then implicit parameter inference takes place within a monomorphic type system. Other polymorphic languages either have no implicit operations, a predefined set of implicit operations (like equality comparison in ML), or require that the operations be packaged inside second-order mechanisms, like classes or generic abstract types.

Full overload resolution, taking the context type into account, is essential for the selection of implicit parameters. Ada is the only well known language that has such overload resolution, and this component of our algorithm is essentially the same as Ada's. ML has a limited form of overload resolution for builtin operators. Other languages, for example PL/I, Algol 68 (van Wijngaarden 1975) and C++ (Stroustrup 1986), do overload resolution based on parameter types alone.

Conclusions

The static type system presented here achieves greater flexibility than existing polymorphic type systems, primarily by avoiding the packaging of operations into secondary entities like classes. This flexibility is achieved without compromising modularity, through a novel combination of simple information hiding, inferred type parameters, inferred operation parameters, and overloading. While these components have been included individually in previous languages, in combination they yield a type system whose expressiveness exceeds the sum of the components.

Acknowledgements

The authors thank Dennis Vadura and Peter Bumbulis for their careful reading of this manuscript. This work is supported by the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre of Ontario, and Rice University.

References

Reference Manual for the Programming Language Ada, U.S. Department of Defense, ANSI/MIL-STD-1815-A (1983)

Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers – Principles, Techniques, and Tools* , Addison Wesley (1986).

Boehm, H., *Partial polymorphic type inference is undecidable*, Proc. 26th Symposium on Foundations of Computer Science (Oct. 1985), 339-345.

Boehm, H. and Demers, A., *Implementing Russell*, Proc. Sigplan 86 Symposium, in Sigplan Not. 21:7 (1986), 186-195.

Boehm, H., *Type Inference in the Presence of Type Abstraction*, Proc. SIGPLAN 89, in SIGPLAN Not. 24:7 (1989), 192-206.

Cardelli, L. and Wegner, P., *On Understanding Types, Data Abstraction, and Polymorphism*, Computing Surveys 17:4 (Dec. 1985), 471-522.

Cardelli, L., *Typeful programming*, Digital Systems Research Center, Palo Alto (1989)

Cormack, G.V. *Extensions to static scoping*, Proc. Sigplan 83 Symposium, in Sigplan Notices 18:6 (1983), 187-191.

Cormack, G.V., and Wright A.K., *Polymorphism in the compiled language ForceOne*, Proc. 20th Hawaii Conf. on System Sciences (1987), 284-292.

Donahue, J. and Demers, A., *Data Types are Values*, A.C.M. Trans. Prog. Lang. Syst. 7:3 (1985), 426-445.

Futatsugi, K., Goguen, J., Jouannaud, J. and Meseguer J., *Principles of OBJ2*, Proc. 12th Symposium on Principles of Programming Languages (1985), 52-66.

Girard, J., *Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur*, These d'Etat, Universite Paris VII, 1972.

Kaes, S., *Parametric overloading in polymorphic programming languages*, Lecture Notes in Computer Science 300, Springer (1988), 131-144.

MacQueen D., *Modules for Standard ML*, Conf. Record of ACM Symposium on Lisp and Functional Programming (1984), 198-207.

Meyer, B., *Object Oriented Software Construction*, Prentice Hall (1988).

Milner, R., *A Theory of Type Polymorphism in Programming*, J. Computer Syst. Sci. 17 (1978), 348-375.

Milner, R., *The Standard ML Core Language*, Polymorphism 2:2 (Oct. 1985), 1-28.

Mitchell, J.C., *Polymorphic type inference and containment*, Information and Computation 76 (1988), 211-249.

Meyer B., *Object-oriented Software Construction*, Prentice Hall (1988).

O'toole, J and Gifford, D., *Type reconstruction with first-class polymorphic values*, Proc. Sigplan 89 Symposium, in Sigplan Not. 24:7 (1989), 207-217.

Reynolds, J., *Towards a Theory of Type Structure*, Paris Colloquium on Programming (1974), 408-424.

Robinson, J.A., *A machine-oriented logic based on the resolution principle*, J. ACM 12:1 (1965), 23-41.

Stroustrup, B., *The C++ Programming Language, Addison Wesley (1986).*

van Wigngaarden, A., et al. *Revised report on the algorithmic language Algol 68*, Acta Informatica 5 (1975), 1-236.

Wadler, P and Blott, S., *How to make ad-hoc polymorphism less ad hoc*, Proc. 16th Symposium on Principles of Programming Languages (Jan. 1989), 60-76.

Wright, A.K., *Design of the Programming Language ForceOne*, Research Report CS-87-10, University of Waterloo (1987).