Parameterized Types and Polymorphic Functions in SIMPL

Muhammed F. Mudawwar
mudawwar@aucegypt.edu
The American University in Cairo
113 Kasr el Aini, Cairo, Egypt

Abstract

Parametric polymorphism or generic programming has been supported by a number of programming languages. Templates and generic parameters are most commonly used. When instantiated, templates and generic units generate new functions, types, packages or modules. This paper shows a different approach to parametric polymorphism in the frame of a new programming language called SIMPL. Polymorphic functions in SIMPL carry two lists of parameters: the hidden parameters and the formal parameters. The hidden parameters, which are used to parameterize a function, appear in the types of the formal function parameters, the result type, and in the body of a function. Hidden parameters are never passed explicitly when a function is called, but rather inferred implicitly by the type system. With hidden parameters, polymorphic functions can have a unique and efficient translation as will be shown in this paper.

Keywords: Parameterized types, polymorphic functions, hidden parameters, SIMPL.

1 Introduction

Programming languages are classified, according to the uniqueness of the type of objects and methods, into monomorphic languages and polymorphic ones. A monomorphic language is the one whose variables and functions can be interpreted to be of one and only one type. In contrast, a polymorphic language is one in which some variables and functions have more than one type. Polymorphism is classified into *universal polymorphism* and *ad-hoc polymorphism*. Universal polymorphism is further classified into *parametric* (*generic*) *polymorphism* and *inheritance polymorphism*. Ad-hoc polymorphism is classified into *overloading* and *coercion* [Cardelli 85].

This paper is about parametric polymorphism. A function is *parametrically polymorphic* if it follows the same algorithm regardless of the type of its arguments. Examples of parametric polymorphism are the Ada generic mechanism [Skansholm 97], the Modula-3 generic mechanism [Cardelli 92], the C++ templates [Stroustrup 97, 88], and the ML type schemes [Milner 89].

This paper introduces the concept of hidden parameters in polymorphic functions. This concept is discussed in the frame of a new programming language called SIMPL (Simplified Imperative Modular Parallel Language). SIMPL is a new programming language, designed mainly to simplify parallel programming on a wide variety of architectures. A polymorphic function in SIMPL has two lists of parameters: the hidden parameters and the formal function parameters. The hidden parameters are specified in the function header, but are not passed explicitly by the programmer. Instead, the type system infers their values when matching the types of the actual parameters against the types of the formal parameters and passes them implicitly for each function call.

The basic problem that must be solved for polymorphic languages is that the compiler cannot know the type and size of the data held in polymorphic variables. Appel discusses four different techniques that are used to solve this problem. These are: *inline expansion, full boxing, coercion-based representation analysis,* and *type-passing* [Appel 98]. Inline expansion is the expansion of polymorphic functions and types until everything is monomorphic. This is the way that Ada generics and C++ templates work. The advantage of this approach is a simple and efficient compilation model. The disadvantage is that functions are replicated, which can cause code bloat.

The full boxing technique fits all polymorphic variables in one representation. Usually, a single word is used with all variables. If a variable is too large to fit in a single word, we allocate a record for that variable and use the pointer to that record as the word. The record representing the boxed variable usually starts with a descriptor indicating how large the record is. The problem with full boxing is that the entire program uses expensive boxed representations, even in places where the programmer has not used any polymorphic functions. The idea of coercion-based representation analysis is to use unboxed representations for values held in monomorphic variables, and boxed (or tagged) representations for values held in polymorphic variables. That way the monomorphic parts of the program can be very efficient, with a price paid only when polymorphic functions are called. Cardelli describes a fully boxed implementation of ML [Cardelli 84], Leroy describes coercion-based representation analysis [Leroy 92], and Shao and Appel describe a variant that does recursive wrapping only when necessary [Shao 95].

Another approach to the implementation of polymorphism is to pass to a polymorphic function a description of the type of polymorphic variables. Harper and Morrisett describe type-passing style [Morrisett 95].

This paper is organized as follows: Section 2 discusses parameterized types in SIMPL. Section 3 discusses polymorphic functions and hidden parameters. Section 4 discusses the unification algorithm used to infer the hidden parameters. Section 5 discusses the translation of polymorphic functions, and Section 6 concludes this paper.

2 Parameterized Types

Types in SIMPL can be parameterized. This type parameterization allows the development of *polymorphic functions* with *hidden parameters* as will be discussed later. A *type declaration* informs the compiler of the existence of a new type. This type may or may not be parameterized. A non-parameterized type is simply an identifier, while a parameterized type should include at least one formal type parameter surrounded by braces. Here are some examples of type declarations:

```
type complex                          -- Not parameterized
type list{t:type}                     -- Parameterized
type array{n:integer,t:type}          -- Parameterized
```

A *formal type parameter* must be either an integer parameter or a type variable. A *type variable* is a formal type parameter of type **type**. The keyword **type** denotes the set of all types declared by a programmer. Thus, a type variable is an unspecified element of **type**. In the above examples, the formal type parameter *t* is a type variable. Limiting a formal type parameter to an integer or to a type variable is not really a restriction. To declare a parameterized type, we need to generalize the size and the type of its elements. The size is generalized using one or more integer parameters. The type is generalized using one or more type variables.

2.1 Type Interface and Type Implementation

A SIMPL *program* is a collection of interfaces and modules stored in separate files and compiled separately. An *interface* describes the interface of a module. It consists mainly of type and function interfaces. A *module* describes the implementation of an interface. It consists mainly of type and function implementations. All interface members are public and exportable to other interfaces and modules. Module members not declared in an interface are private and visible only within the given module.

A *type interface* contains the public members of a type. A type interface is usually specified in an interface file to make it exportable, but it can also be made local to a module. An example of a type interface is shown below. It specifies the interface of the parameterized type *stack*. There are three public member functions, but no public field in this type interface.

Example of a Type Interface:

```
interface stack is

  type stack{n:integer,t:type} is
    function push(x:t)
    function pop():t
    function items():integer
  end  -- interface of type stack

end  -- interface stack
```

A *type implementation* contains the private fields and the implementation of the public functions of a type. It may also contain additional private functions. A type implementation must be specified in a module. It is not exportable and cannot be specified in an interface. The implementation of the parameterized type *stack* is shown below. It begins with the **private** keyword and contains two private member objects: *top* and *storage*, and the implementation of the three public functions: *push*, *pop*, and *items*.

Example of a Type Implementation:

```
module stack is

  private stack{n:integer,t:type} is

    obj top :integer := 0
    obj storage :array{n,t}

    function push(x:t) is  ...  end        -- implementation of push()
    function pop():t is  ...  end          -- implementation of pop()
    function items():integer is  ...  end  -- implementation of items()

  end  -- implementation of type stack

end  -- module stack
```

The implementation of the parameterized type *stack* uses an array of size *n* and element type *t* for internal storage, where *n* and *t* are the formal type parameters. Furthermore, *n* and *t* can be used in the implementation of the member functions. The actual values of *n* and *t* are specified when *stack* objects are declared. The type *stack* is an example of a *polymorphic type* whose size and shape vary according to the actual values of *n* and *t*. The member functions of *stack* are also *polymorphic*, because they can be called with *stack* objects of any size *n* and element type *t*.

2.2 Type Overloading

A type is *overloaded* if a new type with the same identifier has been declared. A type can be overloaded if and only if the new type (with the same identifier) has a different signature. A *type signature* consists of the type identifier and the signature of the formal parameters, which can be either integers or type variables. The signature of an integer parameter is "I", and the signature of a type variable is "T". For example, the following stack types can coexist because their signatures are different. The programmer can implement each stack type differently.

Declared Type:	*Type Signature:*
`type stack`	`stack`
`type stack{t:type}`	`stackT`
`type stack{n:integer, t:type}`	`stackIT`
`type stack{t:type, n:integer}`	`stackTI`

2.3 Type Instantiation and Subtyping

Given a parameterized type, we can derive instances of that type. A *parameterized type instance* is a type identifier followed by a list of actual type parameters. An *actual type parameter* can be an integer identifier, an integer literal, a type variable, a type identifier, or a parameterized type instance. If the formal type parameter is an integer parameter then the actual parameter should be an integer identifier or an integer literal. If the formal type parameter is a type variable then the actual type parameter can be either a type variable, a type identifier, or a parameterized type instance.

Different instances of a parameterized type are considered different types. For example, the types *array*{100 ,*real*}, *array*{200, *real*}, and *array*{100, *array*{100, *real*}} are different. The types *array*{*m*, *real*} and *array*{*n*, *real*} are also considered different even when the values of *m* and *n* turn out to be identical at runtime. This is because type checking is done statically at compile time and the compiler cannot, in general, verify that two integer identifiers have identical values. Finally, integer identifiers used as actual type parameters should not be modifiable objects. They can be integer constants declared in a function, read-only value parameters or hidden parameters of a function as will be clarified later.

A type is said to be *polymorphic* if it contains at least one type variable or an integer parameter. Otherwise, it is called *monomorphic*. For example, the types *t*, *array*{*n*, *array*{*n*, *real*}}, and *stack*{5, *t*} are polymorphic, while the types *integer* and *array*{10, *real*} are monomorphic. Types are also related by instantiation. A subtyping relationship is defined between a polymorphic type and its instances. For example, the types *array*{100, *t*}, *array*{*n*, *real*}, and *array*{100, *array*{200, *real*}} are all subtypes (because they are instances) of *array*{*n*, *t*}. However, the types *array*{100, *t*} and *array*{*n*, *real*} are not related (neither one is a subtype of the other), but have a common subtype which is *array*{100, *real*}.

The subtyping relation defined by type instantiation is a reflexive, anti-symmetric, and transitive relation. The notation <: means a subtype (adopted from [Cardelli 97]). If T is a type then $T <: T$. If $T1 <: T2$ and $T2 <: T1$ then $T1$ and $T2$ must be equivalent. If $T1 <: T2$ and $T2 <: T3$ then $T1 <: T3$.

2.4 Processing Types

A parameterized type in SIMPL is not a type generator as is the case with the class template of C++. A parameterized type is itself a type and occupies a unique place in a symbol table when processed. To process a parameterized type, we need to record its size, its fields, and its member functions. Figure 1 shows a symbol table with entries for the *integer* type, the *arrayIT* type, and the *stackIT* type of Section 2.1. Types and fields are recorded in this symbol table. Functions are omitted here and will be covered in Section 3.

Symbol table entries have a simple structure. Each symbol table entry includes a *name* and an *attribute*. The *name* field can be implemented as a reference to a string in a string space (not shown), while the *attribute* field is a reference to an attribute entry in an attribute space. The *attribute space* is an array of *tags* and *values*. Type signatures are entered in the symbol table rather than type names because they are unique and encode the signature of the formal parameters (using the I and T letters to denote an integer formal type parameter and a type variable respectively). SIMPL identifiers are not case sensitive. They are converted to lowercase when entered in the symbol table.

Each type (built-in or user-defined) has a unique place in the attribute space, whose tag is **type** and whose value is the number of formal type parameters. For example, the type *integer* has zero parameters while the type *arrayIT* has two parameters. If a type is parameterized, the next entries in the attribute space capture the formal parameters. For example, the first parameter of *arrayIT* is an **intpar** (i.e., an integer parameter), and the second formal parameter is an **typevar** (i.e., a type variable). The address (or index) of a type in an attribute space uniquely identifies the type and serves the purpose of specifying the types of objects, fields, parameters, and function results.

Field names are postfixed when entered in a symbol table. The postfix *@stackIT* represents the address (or index) of the type *stackIT* in the attribute space. This encoding will make a field name unique and will simplify its search. A **field** entry in an attribute space holds the address of its type. If the type of a field entry is a subtype of a parameterized type, new entries in the attribute space are allocated for the subtype. A **subtype** entry holds the address of a parameterized type. The entries that follow a **subtype** entry specify the actual parameters of the subtype. The actual parameters to the *array*{*n*, *t*} type of the field *storage* are references to the formal parameters of the type *stackIT* as indicated by the **ref** entries in the attribute space.

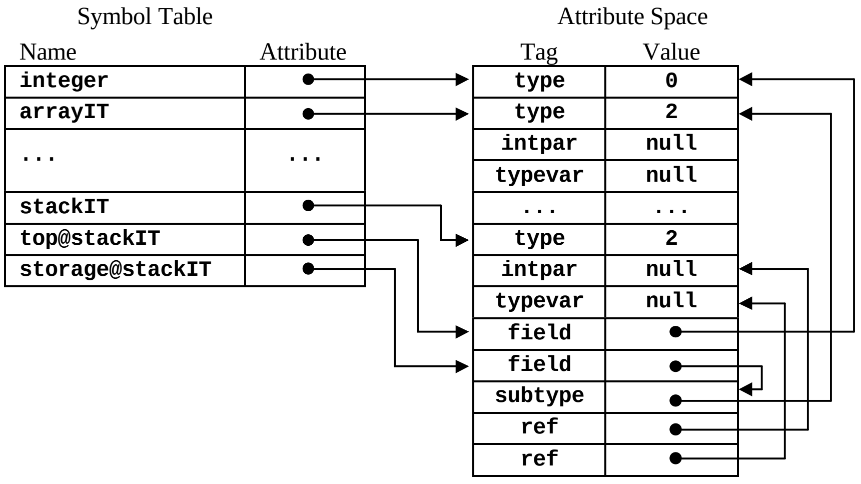

Figure 1 : Symbol table and attribute space showing the type `stackIT` and its fields.

The size of a parameterized type and the offset (relative address) of a field need not be constant values, but are in general functions of the formal type parameters. Furthermore, even when a type is not parameterized and its size is a constant value, the size is not known, in general, when a type is imported because a type interface does not list the private members. Therefore, for each type, the compiler generates a function to calculate its size instead of recording the size in the attribute space. These functions can be expanded inline whenever possible to avoid the overhead of calling them.

The name of a generated *size function* should be encoded in a special way to distinguish it from other functions. Thus, *SIZE@integer*, *SIZE@arrayIT*, and *SIZE@stackIT* are the names of the functions that return the size of the types *integer*, *arrayIT*, and *stackIT*. The address of a type is encoded in the *SIZE* function name. The parameters of a size function are the formal parameters of the corresponding type. The size of a type can be related to the size of other types and can have only addition and multiplication operators when generated. Knowing the size of types, we can determine the size of objects, fields, and parameters, as well as the relative offsets of fields and local declarations. However, the size value is not known in general until runtime and cannot be assumed at compile time.

```
function SIZE@integer():integer is
  result := 4
end -- SIZE@integer

function SIZE@arrayIT(n,SIZEt:integer):integer
  if n <= 0 then result := 0
  else result := n*SIZEt end -- if
end -- SIZE@arrayIT

function SIZE@stackIT(n,SIZEt:integer):integer
  result := SIZE@integer() + SIZE@arrayIT(n,SIZEt)
end -- SIZE@stackIT
```

3 Polymorphic Functions and Hidden Parameters

A *function* denotes a computation that may or may not have a result value. In SIMPL, we do not distinguish between a function and a procedure. All methods are functions. A function with no result value is equivalent to a procedure in other languages, but the keyword **procedure** is not used in this language. Functions in SIMPL can be *polymorphic*. The formal function parameters may assume different types. Polymorphic functions have *hidden parameters*; a feature, I believe, is unique to this programming language.

A *function declaration* is the interface of a function. It informs the compiler about the *formal parameters*, the *hidden parameters*, and the *result type* of a function. A function can be declared in an interface or in a module. However, it cannot be declared twice. A function declared in an interface is public and exportable to other compilation units. A function declared in a module is private and visible only inside the module.

Examples of function declarations are shown below. The first function defines the addition of two complex numbers. This function is *monomorphic* because its actual parameters can assume only one type. The second function is *polymorphic* and uses two lists of parameters. The first list is the *hidden function parameters*, while the second list is the *formal function parameters*. Hidden parameters are formal type parameters used in the types of the formal function parameters, and possibly in the result type of a function. For example, the second function declares *n* as a hidden parameter, and uses it in the types of the formal parameters *x* and *y* as well as in the result type. The third function has two hidden parameters, *n* and *t*, and one formal parameter *x*. Hidden parameters are enclosed by braces, while formal function parameters are enclosed by parenthesis. The **obj** keyword indicates that *x* is an *object* parameter. Formal function parameters are explained next.

Examples of Function Declarations:

```
function add (x,y:complex):complex
function add {n:integer}(x,y:array{n,real}):array{n,real}
function reverse {n:integer,t:type}(obj x:array{n,t})
```

3.1 Formal Function Parameters

A formal function parameter has a mode and a type. The *mode* can be *object* or *value*. The presence of the **obj** keyword means an *object* parameter, while its absence means a *value* parameter. The semantics of formal function parameters are summarized in Figure 2. If the formal parameter is a *value* parameter**,** the actual parameter can be an expression, but if it is an *object* parameter, the actual parameter should be an object. A function body can read *value* and *object* parameters but can write only *object* parameters. A *value* parameter can be passed only to a *value* parameter, while an *object* parameter can be passed either to a *value* or to an *object* parameter.

Formal Parameter	*value*	*object*
Can be read	✓	✓
Can be written		✓
Can be passed to a value parameter	✓	✓
Can be passed to an object parameter		✓

Figure 2: Semantics of formal function parameters

Object parameters are passed by reference, while value parameters are passed *by constant value*. Because no change will occur to a value parameter inside the body of a function, two implementations are possible. The actual parameter can be passed either by value or by reference. For instance, scalar values are passed by value, while large data structures are passed by reference to avoid copying them. However, copying may sometimes be necessary and can be handled transparently by the compiler when the same object is passed to a *value* and to an *object* parameter.

3.2 Hidden Parameters

Hidden parameters are formal type parameters, which are optionally specified in a function declaration. With hidden parameters, a function has two sets of parameters: the hidden parameters and the formal function parameters. Hidden parameters are enclosed by braces and are listed first. Formal parameters are enclosed by parenthesis.

An example of a function with hidden parameters is shown below. The function swap is a polymorphic function that exchanges two rows of a square matrix *m* of any size *n* and element type *t*, where *n* and *t* are hidden parameters. We can access each row in matrix *m* by indexing the first dimension. For example, *m*[*i*] and *m*[*j*] are the *i*th and *j*th rows, respectively.

```
function swap {n:integer,t:type}(obj m:array{n,array{n,t}}, i,j:integer) is
  obj row:array{n,t}
  -- swap row i with row j
  row  := m[i]
  m[i] := m[j]
  m[j] := row
end -- function swap
```

A hidden parameter must be used in the type of at least one formal function parameter, or else it cannot be inferred. Furthermore, a hidden parameter may be used in the body of a function as if it were a formal value parameter. Hidden and formal value parameters are considered constants inside the body of the function and can be used as actual parameters to parameterized types. The function *swap* uses *n* and *t* in the declaration of the local variables.

As the name indicates, hidden parameters are truly hidden. They are not passed explicitly when a function is called, but rather inferred by the type system when *unifying* the types of the actual parameters against the types of the formal parameters. For example, consider the function *CallSwap* with calls to the above function *swap*. Observe that the number of actual function parameters matches the number of formal function parameters, and that the actual values of the hidden parameters are not passed explicitly. In the first call, the inferred value of *n* is 10 and *t* is *real*. In the second call, the inferred value of *n* is 20 and *t* is *array*{10, *real*}. These values are obtained by unifying the types of the actual parameters against the types of the formal parameters. The third call results in a compile-time error since the type system is unable to unify the type of the first actual parameter against the type of the first formal parameter. Matrix *c* has a different number of rows and columns.

```
function CallSwap() is
  obj a:array{10,array{10,real}}
  obj b:array{20,array{20,array{10,real}}}
  obj c:array{10,array{20,real}}
  swap(a,1,3)
  swap(b,0,4)
  swap(c,2,5)  -- compile-time error
end
```

Hidden parameters are needed because they facilitate the development of polymorphic functions, such as the above *swap* function. If hidden parameters were not allowed then additional formal parameters would be required. For example, without hidden parameters the header of the function *swap* would be written differently as shown below. When calling the new *swap* function, we now have to pass actual values for the formal parameters *n* and *t*. Not only is this superfluous and awkward, it is also illegal. SIMPL does not permit the use of type parameters as formal function parameters. The declaration of the new *swap* function results in a compile-time error. Thus, SIMPL forces you to use hidden parameters when declaring or implementing polymorphic functions.

```
function swap(n:integer, t:type, obj m:array{n,array{n,t}}, i,j:integer) is
  -- same as the body of the above swap function
end -- function

function CallSwap() is
  obj a:array{10,array{10,real}}
  swap(10,real,a,1,3)
end
```

3.3 Processing Polymorphic Functions

Parameterized polymorphic functions in SIMPL are not function generators such as the template function of C++. Each function (whether monomorphic or polymorphic) has a unique translation and a unique place in the symbol table and attribute space. This section presents the processing of function headers. Section 5 discusses the translation of polymorphic functions. Consider now the *swap* function of the previous section. This function has the following header:

```
function swap {n:integer,t:type}(obj m:array{n,array{n,t}}, i,j:integer)
```

The internal representation of the *swap* function header is shown in Figure 3. A **function** entry in an attribute space specifies the number of hidden parameters. In this case there are two hidden parameters. The first hidden parameter is an **intpar** (integer parameter) while the second one is a **typevar** (type variable). Following the hidden parameters are the formal function parameters. The formal parameters are collectively called a *tuple*. A *tuple* is an ordered list of elements of possibly different types. A **tuple** entry in an attribute space specifies the number of elements. In this example there are three elements (or formal parameters). The first formal parameter is an **obj** (object parameter) while the second and third parameters are **val** (value parameters). Each parameter entry (whether **obj** or **val**) specifies its type. If the type of a formal parameter is an instance of a parameterized type then a **subtype** entry is allocated in the attribute space. The **subtype** entry points to a parameterized type. The entries that follow the **subtype** entry specify the actual type parameters. The **result** entry in an attribute space specifies the result type of a function. If a function does not have a result type, a null pointer is used as the value of the **result** entry.

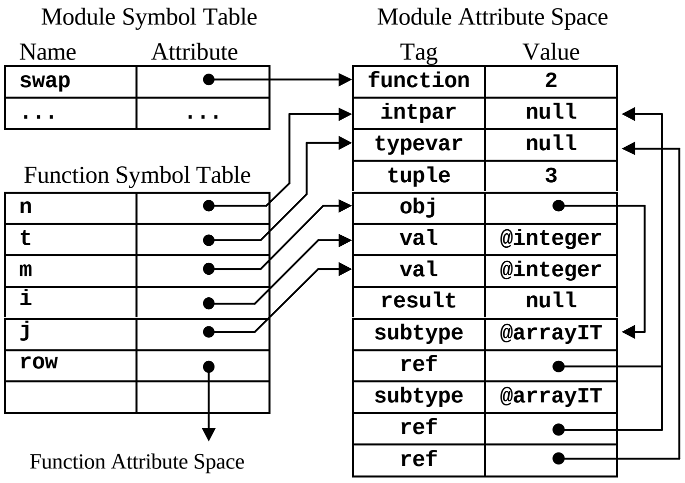

Figure 3: Internal representation of the swap function header

The symbol table of the *swap* function is also shown in Figure 3. Identifiers local to a function (hidden parameters, formal parameters, local object and constant declarations) are placed in a function symbol table. In Figure 3, the hidden parameters *n* and *t*, the formal parameters *m*, *i*, and *j*, and the local object *row* of the function *swap* are placed in this function symbol table. The attributes of the hidden and formal parameters are in the module attribute space, while the attribute of the local object *row* is placed in the function attribute space (not shown in Figure 3). A function symbol table and its attribute space are recycled when the compiler finishes the translation of a function.

Member functions declared in type interfaces are processed exactly like non-member ones. Consider the interface of the type *stack* of Section 2.1. This interface can be written differently as shown below. The two interfaces are semantically equivalent. The difference is syntactic only. Functions encapsulated in a parameterized type use the formal type parameters as hidden parameters. These hidden parameters are manifested when the function is declared outside the type interface. The first formal parameter of a function is omitted when that function is encapsulated in a type interface. The first formal parameter of an encapsulated function is always an **obj** parameter whose type is the encapsulating type.

```
interface stack is

type stack{n:integer,t:type} is
  function push(x:t)
  function pop():t
  function items():integer
end  -- interface of type stack

end  -- interface stack
```

```
interface stack is

type stack{n:integer,t:type}

function push {n:integer,t:type}(obj s:stack{n,t},x:t)
function pop  {n:integer,t:type}(obj s:stack{n,t}):t
function items{n:integer,t:type}(obj s:stack{n,t}):integer

end  -- interface stack
```

3.4 Basic Polymorphic Types, Operators, and Functions

Every programming language needs some basic types, operators, and functions upon which other types and functions can be constructed. The SIMPL language defines four scalar non-parameterized core types and three parameterized ones. The four scalar types are *integer* (32-bit signed), *real* (64-bit IEEE 754-1985 double-precision floating-point standard), *char*, and *boolean*. In practice, we may need more than one integer and floating-point types for space efficiency reasons. The *char* and *boolean* types are called *enumeration types*. All enumeration literals are enclosed by single quotes to distinguish them from identifiers. The enumeration literals for the *boolean* type are `'true'` and `'false'` (enclosed by single quotes). String literals are enclosed by double quotes. Enumeration and string literals are case sensitive, while identifiers and keywords are not.

In addition to the four basic scalar types, SIMPL defines three parameterized core types: the reference type *ref*{*t*}, and the two array types *array*{*n*, *t*} and *array*{*t*}. The reference type has a single parameter that specifies the type of the target object being referenced. A reference object carries either the **null** address, or the address of a dynamically allocated object. The statically sized *array*{*n*, *t*} has two formal parameters that specify the length of the array and the type of its elements. All array objects start at index zero. Array objects of length zero are allowed. Array objects of a negative length are rounded to zero. The dynamically sized *array*{*t*} has a single parameter that specifies the array element type. The length is not specified. The *new* function is used to allocate a dynamic array object of a given length.

Polymorphic functions require the existence of some basic polymorphic operators. SIMPL assignment and equality operators are polymorphic and can be applied to any type *t*. Their interfaces are shown below. The assignment operator copies all the bytes of an expression to an object of the same type *t*. The equality operators (= and <>) compare two expressions byte by byte as long as their types are identical.

```
function := {t:type}(obj object:t, expression:t)
function =  {t:type}(expr1,expr2:t):boolean
function <> {t:type}(expr1,expr2:t):boolean
```

The *new* functions shown below are also examples of core polymorphic functions. The first *new* function is used to allocate a dynamic object of type *t* and returns its address of type *ref*{*t*} through the object parameter *r*. The second *new* function is used to allocate a dynamic array of length *n*. A reference to the dynamic array as well as its length *n* are then stored in the object parameter *a*.

```
function new {t:type}(obj r:ref{t})
function new {t:type}(obj a:array{t}, n:integer)
```

4 Static Type-Checking and Inferring Hidden Parameters

The body of a polymorphic function consists of object declarations, statements, and expressions. When translated, local object declarations allocate storage on the run-time stack. Non-control statements and expressions are calls to their respective functions and operators. For example, the translation of the following assignment statement involves two calls to the index operator and one call to the assignment operator.

```
m[j] := m[i]               Temp1 <- INDEX(m,i)
                           Temp2 <- INDEX(m,j)
                           ASSIGN(Temp2,Temp1)
```

To statically type-check the body of a polymorphic function is equivalent to say that all the calls generated by the statements and expressions are valid calls. A function call is valid if we can unify the types of the actual parameters against the types of the formal parameters. Therefore, the translation of a polymorphic function boils down to calling a polymorphic function and inferring the hidden parameters.

The unification algorithm used to check the validity of a function call and inferring the hidden parameters is shown below. The *unify* function has two parameters: *formal* and *actual* which represent the formal and actual parameter types of a function. The type *attref* (*att*ribute *ref*erence) represents the address (or index) of an attribute entry in an attribute space. Thus, *formal* and *actual* are the addresses of attribute entries. The following functions are defined on attribute references and are used in the *unify* function. The value *v* that can be stored in an attribute entry must be either an integer value or a reference to an attribute entry.

tag(*formal*),	Get the tag of the attribute entry pointed to by *formal*.
value(*formal*),	Get the value of the attribute entry pointed to by *formal*.
setvalue(*formal*, *v*),	Set the value of the attribute entry pointed to by *formal* to *v*.
formal+1	The address of the attribute entry that follows the entry pointed to by *formal*.

The *unify* function returns a boolean value. If it returns 'true' then successful unification was achieved. Otherwise, unification is unsuccessful and the call cannot be made. As a side effect, the *unify* function defines the hidden parameters. The *null* values of the hidden parameter entries are substituted with references to attribute entries that define the actual values of hidden parameters for a given function call.

```
function unify(formal,actual:attref):boolean is
  obj i:integer := 1

  if tag(formal)='ref' then result := unify(value(formal),actual)
  elsif tag(actual)='ref' then result := unify(formal,value(actual))
  elsif tag(formal)='tuple' and tag(actual)='tuple' and value(formal)=value(actual) then
    result := 'true'
    while result and i <= value(formal) do
      result := unify(formal+i,actual+i)
      i := i+1
    end -- while
  elsif tag(formal)='obj' and tag(actual)='obj' then
    result := unify(value(formal,value(actual))
  elsif tag(formal)='val' and (tag(actual)='val' or tag(actual)='obj') then
    result := unify(value(formal),value(actual))
  elsif tag(formal)='typevar' and value(formal)='null' and
       (tag(actual)='type' or tag(actual)='subtype' or tag(actual)='typevar') then
    setvalue(formal,actual)
    result := 'true'
  elsif tag(formal)='typevar' and value(formal)<>'null' then
    result := unify(value(formal),actual)
  elsif tag(formal)='intpar' and value(formal)='null' and
       (tag(actual)='intlit' or tag(actual)='intpar') then
    setvalue(formal,actual)
    result := 'true'
  elsif tag(formal)='intpar' and value(formal)<>'null' then
    result := equiv(value(formal),actual)
  elsif tag(formal) = 'subtype' and tag(actual) = 'subtype' and
        value(formal) = value(actual) then
    result := 'true'
    while result and i <= value(value(formal)) do
      result := unify(formal+i,actual+i)
      i := i+1
    end -- while
  elsif tag(formal) = 'intpar' and (tag(actual) = 'iliteral' or
        tag(actual) = 'intpar' or tag(actual) = 'val') then
    setvalue(formal,actual)
    result := 'true'
  elsif tag(formal) = 'intpar' and value(formal) <> 'null' and
        (value(formal) = actual or (tag(value(formal) = 'iliteral' and
         tag(actual) = 'iliteral' and value(value(formal)) = value(actual))) then
    result := 'true'
  else
    result := 'false'
  end -- if
end -- function unify


function equiv(t1, t2: attref):boolean is
  if t1 = t2 then result := 'true'
  elsif tag(t1) = 'subtype' and tag(t2) = 'subtype' and value(t1)=value(t2) then
    result := 'true'
    while result and i<= value(value(t1)) do
      result := equiv(t1+i, t2+i)
      i := i+1
    end -- while
  elsif tag(t1) = 'ref' then
    equiv(value(t1), t2)
  elsif tag(t2) = 'ref' then
    equiv(t1, value(t2))
  elsif tag(t1) = 'iliteral' and tag(t2) = 'iliteral' and value(t1) = value(t2) then
    result := 'true'
  else
    result := 'false'
  end -- if
end -- function equiv
```

The following example illustrates the operation of the unification algorithm. The headers of functions *f* and *g* are first processed and entered in the module attribute space as shown in Figure 4. The local identifiers of function *g* are entered in *g*'s symbol table. The attributes of local objects *b* and *c* are entered in *g*'s attribute space. To apply function *f* on the tuple (*a*,*b*), we first enter this tuple in the attribute space of function *g* (the caller function). The attribute values of the hidden parameters of function *f* are initialized to null.

```
function f{n:integer,t:type}(obj x:array{n,t}, y:t):t

function g{n:integer,t:type}(obj a:array{n,array{n,t}}) is
  obj b:array{n,t}
  obj c:t
  b := f(a,b) -- ok
  c := f(a,c) -- error
end -- callmult
```

The *unify* function is called by passing the address of the formal tuple of function *f* (in the module attribute space) to *formal* and the address of the actual tuple (in *g*'s attribute space) to *actual* as shown in Figure 4. The *unify* function starts by unifying the type of the first actual parameter against the type of the first formal parameter. The result of this unification is that the attribute value of the first hidden parameter of *f* becomes the address of the first hidden parameter of *g* and the attribute value of the second hidden parameter of *f* becomes the address of the **subtype** entry *array*{*n*,*t*} as illustrated by the dashed lines in Figure 4.

The *unify* function then unifies the type of the second actual parameter against the type of the second formal parameter and validates that both are **subtype** *array*{*n*, *t*}. This is required because of the constraint set in function *f* that the type of *y* and the element type of the array *x* should be identical. Therefore, the *unify* function returns 'true' for the first call. However, it returns 'false' for the second call because it is unable to unify the type of actual parameter *c* (**typevar** *t*) against the type of the formal parameter *y* (**subtype** *array*{*n*,*t*}).

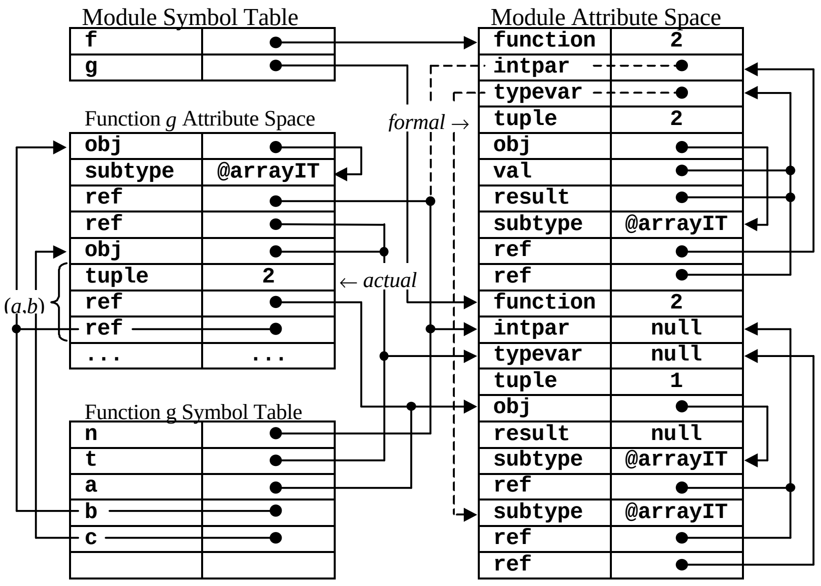

Figure 4: Illustrating the unification algorithm

After a successful unification, we obtain the attribute values of the hidden parameters of the callee function. These values are important for translation. We also get the result type for further use. For example, we need the result type of *f*(*a*, *b*) (which is **subtype** *array*{*n*,*t*}) to validate the call to the assignment operator as well as for translation.

5 Translating Polymorphic Functions with Hidden Parameters

To have a unique translation for a polymorphic function with hidden parameters, we must have a unique translation for the basic polymorphic operators. This is because a polymorphic function, such as the *swap*() function of Section 3.2, is written using basic polymorphic operators, such as the assignment operator. Fortunately, a unique translation for the basic polymorphic operators exists.

Let *SIZEt* be the size of type *t* (in bytes). There can be a unique translation for each of the basic polymorphic operators and functions as long as *SIZEt* is passed as a hidden parameter. The assignment operator needs to know *SIZEt* to determine the number of bytes to be copied. The equality operators (= and <>) need *SIZEt* to determine the number of bytes to be compared. The *new* function needs *SIZEt* to determine the number of bytes to be allocated. The index operator [ ] needs *SIZEt* to calculate the address of the indexed element. Observe that the actual type of *t* is not important in this discussion. *SIZEt* is all that is required. In general, a polymorphic function with a type variable as a hidden parameter, needs to know the *size* of the type variable for each function call. This size, which is equal to the size of the actual parameter, can be obtained for each function call after inferring the hidden parameters.

Consider the translation of the polymorphic assignment and the array index operators. The translated functions *ASSIGN* and *INDEX* are shown below. The hidden parameters become formal parameters of the translated functions. *SIZEt* is passed as an integer parameter. The type *address* means a virtual address in the address space of a process. The parameters of a translated function are normally passed through registers. *Integer* and *address* parameters use integer registers. I distinguish them here for clarity. User-defined types vanish when a function is translated.

```
function ASSIGN (SIZEt:integer, object:address, expression:address) is
  -- copy SIZEt bytes from expression to object
  -- account for the possible overlap between object and expression
end
```

```
function INDEX (n:integer, SIZEt:integer, a:address, i:integer):address is
  if i >= 0 and i < n then
    result := a+i*SIZEt
  else
    Index is Out of Range
  end -- if
end
```

Now consider the translation of the *swap* function of section 3.2. This function has 2 hidden parameters *n* and *t* and 3 formal parameters *m*, *i*, and *j*. All become formal parameters in the translated function. The body of the *swap* function had one local object *row*. The size of *row* is calculated by calling the *SIZE@arrayIT* function with actual parameters *n* and *SIZEt*. This is because *row* is of type *array*{*n*,*t*}. We then allocate space for *row* on the stack by modifying the stack pointer. This modification to the stack pointer is done after the function *swap* is called (i.e., after allocating a frame for the temporaries, saved registers, return address, and other known fixed size local objects). The indexing and assignment operations are then translated into calls to the polymorphic *INDEX* and *ASSIGN* functions. These functions require hidden parameters, which are inferred by the unification algorithm. Observe that actual integer parameters are passed for hidden parameters (which are no longer hidden) in the assignment and the indexing operator calls.

```
function swap(n:integer, SIZEt:integer, m:address, i:integer ,j:integer) is
  row  :address
  Temp1:integer
  Temp2:address
  Temp3:address
  Temp1   <- SIZE@arrayIT(n,SIZEt)   -- size of array{n,t}
  row     <- PUSH(Temp1)             -- modify the stack pointer
  Temp2   <- INDEX(n,Temp1,m,i)      -- compute address of m[i]
  ASSIGN(Temp1,row,Temp2)            -- row := m[i];  copy Temp1 bytes
  Temp3   <- INDEX(n,Temp1,m,j)      -- compute address of m[j]
  ASSIGN(Temp1,Temp2,Temp3)          -- m[i] := m[j]; copy Temp1 bytes
  ASSIGN(Temp1,Temp3,row)            -- m[j] := row;  copy Temp1 bytes
  POP(Temp1)                         -- restore the previous stack pointer
end
```

6 Conclusion and Further Research

This paper has shown the importance of hidden parameters to polymorphic functions. It has also demonstrated that polymorphic functions can have a unique translation, provided that hidden parameters are inferred and passed properly. Type information was not required to translate a polymorphic function. The size of a type variable is all that was required. At this point, I would like to caution the reader that this statement is not always true and type information is sometimes needed at runtime. This is the case if *function specialization* is permitted in a programming language. Let me clarify this point.

This paper has focused only on one kind of polymorphism, namely parametric polymorphism. Neither overloading nor inheritance was discussed to limit my scope. Overloading of polymorphic functions adds a second dimension to a polymorphic language. With overloading, one can overload the assignment operator to allow the assignment of a *real* to a *complex*, or a value of type *t* to all the elements of an array of type *array*{*n*,*t*} as shown below:

```
function := (obj x:complex, y:real)
function := {n:integer,t:type}(obj x:array{n,t}, y:t)
```

A special case of function overloading is called *function specialization*. An overloading function is said to be a specialization of an overloaded function, if the type of the overloading function parameters is a subtype of the overloaded function parameters. For example, one may wish to overload the assignment operator for *list*{*t*} parameters to handle the appropriate copying of lists. This overloading is called specialization because the type of the specialized function parameters (*list*{*t*}, *list*{*t*}) is a subtype of the overloaded function parameters (*t*, *t*). Similarly, the third function is a specialization of both the second and first functions.

```
function := {t:type}(obj x:t, y:t)
function := {t:type}(obj x:list{t}, y:list{t})
function :=         (obj x:list{real}, y:list{real})
```

Overloading with specialization is more powerful in a programming language but much more difficult to implement than overloading without specialization. There are essentially two problems to solve:

- We need to identify and implement the rules for unambiguous overloading.
- We need to bind function calls to function addresses.

An example of ambiguous overloading is given below. If function *f* is called with an actual parameter of type *array*{100, *real*} then both functions can apply. This is because the types *array*{*n*, *real*} and *array*{100, *t*} are not related but have the common subtype *array*{100, *real*}. This ambiguous overloading can be eliminated by either dropping one of the two functions, or by introducing a third function *f* with parameter type *array*{100, *real*}.

```
function f {n:integer}(obj x:array{n,real})
function f {t:type}   (obj x:array{100,t})
```

A more serious problem with function specialization is that we cannot always determine statically at compile time the address of a called function. In other words, it is not always possible to statically bind a function call to a function address. Type information is needed at runtime to achieve dynamic or late binding. The following example clarifies this problem. Function *g* calls function *f*. If *x* is of an *array* type, the second function *f* should be called. Otherwise, the first one should be called. The difficulty here is that the specialized (second) function *f* can be defined in a different module after function *g* has been compiled. Other specialized functions of *f* can be defined elsewhere. Therefore, *SIZEt* is no longer sufficient as a hidden parameter and type information is required when function specialization is permitted in a programming language. The problem of overloading with specialization is currently being investigated and will be addressed in depth in a separate paper.

```
function f {t:type}(obj x:t)
function f {n:integer,t:type}(obj x:array{n,t})
function g {t:type}(obj x:t) is
  ...
  f(x)
  ...
end
```

References

[Appel 98] A. Appel, *Modern Compiler Implementation*, Cambridge University Press, 1998.

[Cardelli 97] L. Cardelli, "Type Systems", *Handbook of Computer Science and Engineering*, Chapter 103, CRC Press, 1997.

[Cardelli 92] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson, "Modula-3 Language Definition", *ACM Sigplan Notices*, vol. 27, no 8, August 1992, pages 15-43.

[Cardelli 85] L. Cardelli and P. Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys*, vol. 17, no 4, pages 471-522, December 1985.

[Cardelli 84] L. Cardelli, "Compiling a Functional Language", *1984 Symposium on LISP and Functional Programming*, 1984, pages 208-217.

[Harper 95] R. Harper and G. Morrisett, "Compiling Polymorphism using Intensional Type Analysis", *22nd ACM Symposium on Principles of Programming Languages*, 1995, pages 130-141.

[Ichbiah 79] J. Ichbiah, et al., "Rationale for the Design of the ADA Programming Language", *ACM Sigplan Notices*, vol. 14, no 6, June 1979

[Milner 89] R. Milner, M. Tofte, and R. Harper, *The definition of Standard ML*, MIT Press, 1989.

[Mudawwar 98] M. Mudawwar, "SIMPL: Language Definition", Technical Report, Computer Science Department, The American University in Cairo, June 1998.

[Leroy 92] X. Leroy, "Unboxed Objects and Polymorphic Typing", *19th ACM Symposium on Principles of Programming Languages*, 1992, pages 177-188.

[Shao 95] Z. Shao and A. Appel, "A type-based compiler for standard ML", *1995 ACM Conference on Programming Language Design and Implementation*, pages 116-129.

[Skansholm 97] J. Skansholm, *Ada 95 From the Beginning*, Addison-Wesley, New York, 1997.

[Stroustrup 97] B. Stroustrup, *The C++ Programming Language*, 3rd edition, Addison-Wesley, 1997.

[Stroustrup 88] B. Stroustrup, "Parameterized Types for C++", USENIX C++ Conference, Denver, October 1988.

[Thompson 95] S. Thompson, *Miranda: the Craft of Functional Programming*, Addison-Wesley, Wokingham, 1995.

[Wirth 85] N. Wirth, "Programming in Modula-2", *Texts and Monographs in Computer Science*, Springer-Verlag, Germany, 1985.