Parameter Transmission Abstractions

M. O. JOKINEN
Department of Computer Science, University of Turku, SF-20500 Turku, Finland

In many programming languages parameter transmission involves implicit actions like copying, type conversions or assigning default values to optional parameters. We propose a linguistic mechanism that allows such actions to be defined within the language. The same mechanism can also be used to define most conventional parameter transmission mechanisms and pattern-matching based parameter binding

Received September 1987, revised May 1988

1. INTRODUCTION

In many programming languages the values of actuals in a procedure call are not passed as such to the subprogram. The values may undergo various conversions before they are bound to formals. Parts of the data objects may be copied. Sometimes the conversion process may involve more than a single formal-actual pair and the number of actuals may be different from the number of formals. There may be optional parameters which get certain default values if omitted in the call, or a single actual may define the value of several formals, as conformant array parameters in Pascal.[1] Implicit actions allow a more compact notation and their proper use may thereby improve readability. Unfortunately the rules are usually built in the language and although modern languages allow the definition of application-specific types, they rarely[2] provide any way to extend implicit actions to user-defined types.

The FEXPR feature of Lisp[3] is one method to give the programmer more control over the actual parameters. The list of actual parameters is passed as such and can be freely manipulated in the called routine. The method relies on the representation of programs as list structures and the existence of a user-callable EVAL function. Another approach to handle optional, repeatable and variable-type parameters has been suggested by Prasad[4] and Ford and Hansche.[5] Their methods include syntax extensions to specify formal and/or actual parameters with such properties, and special statements or standard functions to test the existence of optional parameters, the number of repeatable parameters and the actual type of variable-type parameters. These mechanisms, unlike the FEXPR feature, were designed as extensions to strongly typed languages.

In this paper we present a new linguistic mechanism that allows the programmer to define various actions in parameter transmission. The details of parameter transmission can be hidden not only from the calling program but also from the body of the called routine. In other words, the formal parameter part of a routine is an abstract interface between the caller and callee. Transmission rules can be defined separately from the routines in which they are used. Even libraries of transmission rules could be built.

2. THE LANGUAGE

It is easier to represent the abstraction mechanism in an orthogonal and fully dynamic language. In particular, type checking and binding of identifiers are assumed to occur at run time, and types and procedures are assumed to be first class citizens in the language. At the end of the paper we shall discuss the possibility of embedding the abstraction mechanism into languages with static binding and strong typing.

Here is a sketch of the syntax of the language:

$$
\begin{array}{rl}
e = & \text{literal} \\
| & \text{identifier} \\
| & [e_1, \ldots, e_n] \\
| & \textbf{env}\ (e_{11} = e_{12}, \ldots, e_{n1} = e_{n2}) \\
| & \textbf{use}\ e_1\ \textbf{in}\ e_2 \\
| & \textbf{proc}\ e_1 \Rightarrow e_2 \\
| & e_1\ e_2 \\
| & e_1;\ e_2 \\
| & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \\
| & \textbf{case}\ e_0\ \textbf{in}\ e_{11} \Rightarrow e_{12}, \ldots, e_{n1} \Rightarrow e_{n2} \\
 & \quad\quad \textbf{else}\ e_{n+1,1} \Rightarrow e_{n+1,2} \\
| & \textbf{abort}
\end{array}
$$

The above syntax is written in a semiabstract form. Extra parentheses and indentation are used to disambiguate the parse structure when necessary.

2.1 Classes of data objects

Literals include denotations for integers, strings and other scalars.

Clause $[e_1, \ldots, e_n]$ evaluates to a *tuple*. Tuples are ordered sequences of zero or more data objects. A one-element tuple is *not* identical with its element. Expressions e_i can be evaluated in an arbitrary order, or interleaved.

Clause $\textbf{env}(e_{11} = e_{12}, \ldots, e_{n1} = e_{n2})$ evaluates to an *environment*. An environment is a mapping from a finite set of strings into data objects; it is a generalization of the *record* and *package* concepts. Subexpressions e_{ij} are evaluated in an arbitrary order. Each e_{i1} must evaluate to a string. The resulting environment binds strings e_{i1} to the values of expressions e_{i2}.

An environment can be used in a clause

$\textbf{use}\ e_1\ \textbf{in}\ e_2$

where clause e_1 evaluates to an environment. The value of this clause is the value of e_2 whose free identifiers are bound as in the environment yielded by e_1. Clauses e_1 and e_2 can be evaluated in any order or interleaved; this gives some extra freedom in optimization as will be seen in section 4.

The whole program is implicitly embedded in an environment that contains the definitions of standard identifiers. Standard procedure *select* can be used to find the value of an identifier in an environment. The value of $select[e, x]$ is the value bound to string x in environment e. Both operands can be arbitrarily complex expressions. Procedure *econcat* concatenates two or more environments. Clause $econcat[e_1, \ldots, e_n]$ returns an environment that contains the combination of bindings from environments $e_1, \ldots, e_n$. If an identifier is bound in more than one e_i, its value is taken from the last one.

Environments as first class objects can be traced back to Simula,[6] and they have recently gained new interest in the field of language design.[7,8] Our environments are semantically close to the corresponding structures of the Pebble language,[9] which has considerably inspired the ideas presented in this paper.

The evaluation of the **abort** causes a failure, or an exception, i.e. termination without any result. Failures are propagated so that if a subclause of clause e fails while e is being evaluated, then e fails too. However, failures can be trapped in case-clauses, as will be described in section 2.5.

Components of the serial clause $(e_1; e_2)$ are evaluated from left to right and the value of the clause is the value of e_2. The value of the clause **if** e_1 **then** e_2 **else** e_3 is the value of either e_2 or e_3, depending on the value of the boolean clause e_1.

2.2 Procedures

Before discussing procedure definitions in our language, let us examine more closely the semantics of call-by-value parameter transmission in conventional programming languages. The definition of a procedure usually looks something like

$$p = \mathbf{proc}(x_1; t_1, \ldots, x_n: t_n)e$$

where e is the body of the procedure. The call of this procedure is written as

$$p(e_1, \ldots, e_n)$$

where the result of clause e_i is of type t_i. The effect of the call is that the body e of p is evaluated in an environment in which each x_i is bound to the value of e_i. This call is therefore equivalent with the following use-clause:

$$\mathbf{use\ env}(\text{"}x_1\text{"} = e_1, \ \ldots\ , \text{"}x_n\text{"} = e)\ \mathbf{in}\ e$$

Thus the formal parameter part $(x_1: t_1, \ldots, x_n: t_n)$ can be regarded as a function that maps the tuple $[e_1, \ldots e_n]$ into the environment $\mathbf{env}(\text{"}x_1\text{"} = e_1, \ldots \text{"}x_n\text{"} = e_n)$.

Since environments can be treated as first-class objects, it is natural to attempt a further step and consider also the formal parameter part as an ordinary procedure. Any environment-valued procedure can then be freely used as a formal parameter part of another procedure. In such a language a variety of parameter transmission abstractions can be built from a few elementary formals, as will be demonstrated in section 3.

In our language a procedure object is created with a clause

$$\mathbf{proc}\ e_1 \Rightarrow e_2$$

where e_1 is an arbitrary clause that evaluates to an environment-valued procedure (from now on we shall call all such procedures as generalized formals or simply as formals) and clause e_2 is the body of the procedure. All procedures take exactly one argument and (if the evaluation terminates without a failure) return exactly one result. Multiargument procedures can be reduced to single-argument procedures either by currying or by treating the argument list as a tuple. Parameterless procedures formally take an empty tuple as an argument.

Procedure invocations are written as

$$e_1\ e_2$$

where clause e_1 evaluates to a procedure and e_2 evaluates to its argument. If the value of e_1 is $\mathbf{proc}\,f \Rightarrow b$, the invocation is equivalent to $\mathbf{use}\,f\,e_2\ \mathbf{in}\ b$. Nested invocations associate from left to right; thus clause $(e_1\, e_2\, e_3)$ is equivalent to $((e_1\, e_2)\, e_3)$. For convenience, certain operators will be written in their familiar infix or postfix notation. For instance, we shall write $x := y$ instead of $:=[x, y]$.

2.3 Basic formal generators

The language must contain a set of standard formals or formal generators as elementary building blocks for user-defined procedures. We shall first introduce a procedure named *atomf*, which generates 'atomic' formals. It accepts as an argument a 2-tuple $[s, t]$, where s is a string and t is a type. The value of the clause

$$atomf[s, t]$$

is a procedure that maps an object x (of type t) into an environment that binds s to x. If x is not of type t, the call causes a failure.

$$atomf[s, t]x = \begin{cases} \mathbf{env}(s = x), & \text{if } x \text{ is of type } t \\ \mathbf{abort} & \text{otherwise} \end{cases}$$

Note that s may be an arbitrary string-valued expression and it is the value of s (rather than the identifier s) that becomes bound in the environment. For example,

$$atomf[\text{"}n\text{"}, int]4 = \mathbf{env}(\text{"}n\text{"} = 4)$$

For tuple arguments we first introduce a procedure, denoted by *nullf*, that accepts an empty tuple as its argument and returns an empty environment. Thus

$$nullf[\,] = \mathbf{env}()$$

$$nullf x = \mathbf{abort}, \qquad \text{if } x \neq [\,]$$

Next we introduce a procedure, denoted by *fconcat*, that maps 2-tuples of formals to formals. The value of the clause $fconcat[f_1, f_2]$ is a formal that accepts as its argument a nonempty tuple whose first element is accepted by the formal f_1 and whose tail is accepted by the formal f_2. The result of the concatenated formal is an environment which is constructed by combining the environments yielded by f_1 and f_2.

$$fconcat[f_1, f_2][e_1, \ldots, e_n] = econcat[(f_1\, e_1), f_2[e_2, \ldots, e_n]]$$

$$fconcat[f_1, f_2][\,] = \mathbf{abort}$$

$$fconcat[f_1, f_2]x = \mathbf{abort}, \qquad \text{if } x \text{ is not a tuple}$$

For convenience, we shall often use an additional formal generator *tuplef*, which can be defined in terms of *nullf* and *fconcat*:

$$tuplef[] = nullf$$

$$tuplef[f_1, f_2, \ldots, f_n] = fconcat[f_1, tuplef[f_2, \ldots, f_n]]$$

2.4 Types

Since type checks occur at runtime in our language, there must be a sensible action taken when a type check fails. We define a failing type check equivalent to the execution of **abort**. In the examples to follow we will use standard types *int*, *real*, *string*, *anyenv*, *anytuple*, *any* and *type*, and type constructors **ref**, **union**, **tuple** and $\rightarrow$. Type **ref** t is the type of pointers to t-typed cells. Type **union**$[t_1, \ldots, t_n]$ is a coalesced union of types $t_1, \ldots, t_n$. The value space of a union type is the set-theoretic union of the value spaces of component types. Type **tuple**$[t_1, \ldots, t_n]$ is the type of tuples $[x_1, \ldots, x_n]$ where x_i is of type t_i. Clause $t \rightarrow u$ denotes the type of functions with domain t and range u. Identifier *anyev* denotes the type of all environments, *anytuple* denotes the union of all tuple types and *any* denotes the union of all (nonunion) types. Identifier *type* denotes the type of all types (including or excluding *type*).

Union types (either the **union** constructor or *any*) are essential to the expressive power of the abstraction mechanism. Other types are more or less optional, replaceable by each other, or required only in specific examples.

2.5 Case-clause

The syntax of the case-clause is

$$\textbf{case } e \textbf{ in } f_1 \Rightarrow e_1, \ldots, f_n \Rightarrow e_n \textbf{ else } f_{n+1} \Rightarrow e_{n+1}$$

where the values of clauses f_1 to f_{n+1} are formals. The else-part is optional. The clause is evaluated by first evaluating the clause e and then invoking formals f_1 to f_n (in an unspecified order) using the value of e as the argument. If the invoked formal f_i returns an environment, then the clause e_i is evaluated in that environment and the value of e_i becomes the value of the case-clause. If f_i fails, then the next formal is tried. If all the formals f_1 to f_n fail, then the optional formal f_{n+1} is invoked and the clause e_{n+1} is evaluated in the resulting environment. If f_{n+1} fails too or if there is no else-part, then the case-clause fails.

3. EXAMPLES

3.1 Implicit type conversions

As a simple example, let us define a generator for formals that accept either a real or an integer as their actual argument and convert it into a real in the latter case. Standard procedure *inttoreal* performs the conversion explicitly.

```
intreal =
  proc atomf["id", string] ⇒
    proc atomf["x", union[int, real]] ⇒
      atomf[id, real]
        (case x in
          atomf["r", real] ⇒ r,
          atomf["n", int] ⇒ inttoreal n)
```

Here the case-clause is used to compute the argument of *atomf*[*id*, *real*]. Type **union**[*int*, *real*] could be replaced with the type *any*. Formal (*intreal x*) would be normally used in definitions of arithmetic functions. However, *atomf*[*x*, *real*] could be used instead in cases where an integer argument makes no sense. For example, assume that we need a procedure that computes the integral of a given function f over a closed interval $[a, b]$ in the accuracy *epsilon*. The header of the procedure might look like this:

```
proc tuplef[atomf["f", real → real],
            intreal "a",
            intreal "b",
            atomf["epsilon", real]] ⇒ . . .
```

As an analogous but a more specialized example, let us define a generator for formals that accept as an argument a month represented either as an integer or as a string:

```
proc atomf["id", string] ⇒
  proc atomf["x", union[int, string]] ⇒
    atomf[id, int]
      (case x in
        atomf["n", int] ⇒
          if n < 1 or n > 12
          then abort
          else n,
        atomf["s", string] ⇒
          if s = "January" then 1
          else if s = "February" then 2
          . . .
          else if s = "December" then 12
          else abort)
```

3.2 Optional parameters

Procedures with a variable number of arguments can be constructed by treating the list of arguments as a tuple. One possibility is to define a fixed number of normal arguments and bind the rest of the argument tuple to one identifier. For example, in the following formal the length of the fixed part is one:

```
fconcat[atomf["head", t], atomf["tail", anytuple]]
```

Another possibility is to define optional arguments that get default values if omitted in the call. The following procedure takes a list L of 3-tuples [*name*, *type*, *default_value*] and returns a formal that accepts a tuple A whose ith element corresponds to the ith element of the tuple L. The length of A may be smaller than the length of L, and in that case the missing elements get default values from L.

```
optlist = proc atomf["L", anytuple] ⇒
  case L in
    nullf ⇒ nullf,
    fconcat [tuplef[atomf["name", string],
                    atomf["t", type],
                    atomf["default", any]],
             atomf["tail", anytuple]] ⇒
      proc atomf["A", anytuple] ⇒
        case A in
          nullf ⇒ defaults L,
          fconcat[atomf["x", t],
                  atomf["rest", anytuple]]⇒
          econcat[atomf[name, t]x, optlist tail rest]
```

where

```
defaults = proc atomf["L", anytuple] ⇒
  case L in
    nullf ⇒ env(),
    fconcat[tuplef[atomf["name", string],
                   atomf["t", type],
                   atomf["default", any]],
            atomf["tail", anytuple]] ⇒
      econcat[atomf[name, t]default, defaults tail]
```

If there are many optional parameters it is more convenient to identify them by names than by position. Each optional actual parameter is given as a (sub)tuple [*name*, *value*] in the argument list. The following procedure takes the specification of optional arguments in the same form as above but the resulting formal accepts a list of 2-tuples in an arbitrary order:

```
optset = proc atomf["L", anytuple] ⇒
  proc atomf["T", anytuple] ⇒
    econcat[defaults L, values[types L, T]]
```

Procedure *types* computes an environment that maps the names of the formal arguments to their types. This environment is used in the other auxiliary procedure to check the types of actual arguments.

```
types = proc atomf["L", anytuple] ⇒
  case L in
    nullf ⇒ env(),
    fconcat[tuplef[atomf["name", string],
                   atomf["t", type],
                   atomf["default", any]],
            atomf["tail", anytuple]] ⇒
      econcat[atomf[name, type]t, types tail]

values = proc tuplef[atomf["ttable", anyenv],
                     atomf["T", anytuple]] ⇒
  case T in
    nullf ⇒ env(),
    fconcat[tuplef[atomf["name", string],
                   atomf["value", any]],
            atomf["tail", anytuple]] ⇒
      econcat[atomf[name, select[ttable, name]]
              value),
            values[ttable, tail]]
```

3.3 Pattern-matching formals

Especially in recent years it has become popular to write the formal parameter part as a pattern. A pattern is a data structure in which certain elements denote variables to be bound in an invocation. Patterns can be easily defined in our system. Below is a generator for patterns of possibly nested tuples. Variables are denoted by strings that begin with a capital letter.

```
pattern = proc atomf["p", any] ⇒
  case p in
    nullf ⇒ nullf,
    atomf["s", string]⇒
      if s[1] ≥ 'A' and s[1] ≤ 'Z'
        then atomf[s, any]
        else(proc atomf["t", string] ⇒
               if s = t
                 then env()
                 else abort),
    fconcat[atomf["head", any],
            atomf["tail", anytuple]] ⇒
      fconcat[pattern head, pattern tail]
```

For example, the value of the clause

pattern["*f*", ["*X*", "*Y*"], ["*g*", "*Z*"]]

is a formal that accept all tuples that can be constructed by replacing "*X*", "*Y*" and "*Z*" with any objects in the tuple ["*f*", ["*X*", "*Y*"], ["*g*", "*Z*"]]. Patterns for other data types can be defined in an analogous way.

In a more realistic program the types of the variables would be included in patterns and the formal generator would take care of multiple occurrences of a variable. A quotation mechanism is also desirable to permit arbitrary constant terms in patterns (like strings beginning with a capital letter in this example). These features can be defined in the language without difficulty.

3.4 Transmission mechanisms

Transmission mechanisms are closely related to types. If the type system of the language is rich enough, transmission by various mechanisms can be reduced to transmission of various types of data.[10] Call by reference is equivalent to transmission of a parameter of type **ref** *t*. Call by name is equivalent to transmission of a parameter of type *void* → *t*, where *void* = **tuple**[]. Call by need is equivalent to transmission of a recipe, an object of type **ref union**[*t*, *void* → *t*]. However, the programmer may still want to think in terms of transmission mechanisms rather than in terms of types. To make the underlining type system transparent, an argument should undergo an implicit type conversion when it is transmitted further by a different method.

We shall first define two auxiliary procedures. *Rep_value* generates procedures that compute values of recipes.

```
rep_value = proc atomf["t", type] ⇒
  proc atomf["x", ref union[t, void → t]] ⇒
    case x ↑ in
      atomf["y", t] ⇒ y,
      atomf["f", void → t] ⇒ (x := f[]; x ↑ )
```

Here *x* ↑ denotes the contents of the cell pointed by *x*. The other auxiliary procedure *rcpdefs* just generates two shorthand notations, *rcp* and *u*.

```
rcpdefs = proc atomf["t", type] ⇒
  env("rcp" = ref union[t, void → t],
      "u" = union[t, void → t, ref t, rcp])
```

Call by value, name, need and reference, and all the required conversions, can now be defined with the following procedures.

```
value = proc tuplef[atomf["id", string],
                    atomf["t", type]] ⇒
  use rcpdefs t in
    proc atomf["x", u] ⇒
      env(id =
        casex in
          atomf["y", t] ⇒ y,
          atomf["p", ref t] ⇒ p ↑ ,
          atomf["f", void → t] ⇒ f[],
          atomf["r", rcp] ⇒ rcp_value t r)
```

```
name = proc tuplef[atomf["id", string],
                   atomf["t", type]] ⇒
  use rcpdefs t in
    proc atomf["x", u] ⇒
      env(id =
        case x in
          atomf["y", t]⇒(proc nullf ⇒ y),
          atomf["p", ref t] ⇒ (proc nullf ⇒ p ↑ ),
          atomf["f", void → t] ⇒ f,
          atomf["r", rcp] ⇒
            (proc nullf ⇒ rep_value t r))
          atomf["f", void → t] ⇒ f,
          atomf["r", rcp] ⇒
     (proc nullf ⇒ rep_value t r))
need = proc tuplef[atomf["id", string],
                   atomf["t", type]] ⇒
  use rcpdefs t in
    proc atomf["x", u] ⇒
      env(id =
        case x in
          atomf["y", t] ⇒ new rcp y,
          atomf["p", ref t] ⇒ new rcp (p ↑ ),
          atomf["f", void → t] ⇒ new rcp f,
          atomf["r", rcp] ⇒ r)
```

where clause (*new rcp e*) allocates a new cell of type *rcp*, initializes its contents to *e* and returns a pointer to the cell.

```
reference = proc tuplef[atomf["id", string],
                        atomf["t", type]] ⇒
  use rcpdefs t in
    proc atom["x", u] ⇒
      env(id =
        case x in
          atomf["y", t] ⇒ new t y,
          atomf["p", ref t] ⇒ p,
          atomf["f", void → t] ⇒ new t(f[]),
          atomf["r", rcp] ⇒ new t(rcp_value t r))
```

Call by result cannot be implemented in this way because it involves implicit actions at the termination rather than at the start of the called procedure.

4. IMPLEMENTATION

The above language has not yet been implemented, but in this section we will briefly discuss the compilation of generalized formals in conventional computers. The text below is an introduction of implementation ideas rather than a comprehensive analysis, but we try to demonstrate that in many cases there is a natural way to compile the formals into efficient instruction sequences.

In a stack-oriented machine a procedure call $p(e_1, \ldots, e_n)$ is compiled into the following code:

$code(e_1)$
$conv(e_1)$
$code(e_2)$
$conv(e_2)$
...
$code(e_n)$
$conv(e_n)$
$code(p)$

Code(e_i) is an instruction sequence that evaluates the expression e_i and leaves its value on the top of the stack. *Conv*(e_i) is a (possibly empty) instruction sequence that converts the primary value of e_i into the type required by *p*. *Conv*(e_i) replaces the value of e_i on the stack with the converted value. *Code*(*p*) computes the value of the clause *p* (unless it can be evaluated at compile time) and transfers control to the body of the procedure. Values of arguments form the lower part of the activation record of the procedure. The overall effect of the execution of the body of *p* is that the stacked values of the arguments are replaced by the result of the routine.

In our language the invocation (**proc** *f* ⇒ *e*) *a* is, by definition, equivalent to **use** *f a* **in** *e*, which can be compiled into the following instruction sequence:

code(*a*)
code(*f*)
code(*e*)

Code(*a*) leaves the value of the argument *a* on the top of the stack. *Code*(*f*) replaces it with an environment yielded by the formal *f*. *Code*(*e*) may be either an inline instruction sequence for *e* or a jump to the start of the instruction sequence for *e* (which must then be terminated by a return-instruction). The latter alternative is more natural for use-clauses originated from invocations. The result of *f* is used to bind the free identifiers of *e* – how that is done depends on the representation of environments.

General environments can be represented as association lists, hash tables, binary trees or combinations of these (and possibly other) structures. However, in the special case that the bound identifiers are known at compile time, an environment can be represented exactly like a conventional record: the components of the environment can be stored in consecutive memory locations and the value of an identifier is found by adding a static offset to the base address of the environment. In this case the environment produced by a formal can be used as the lower part of the activation record of a procedure as in conventional languages. The order of the components is irrelevant but it should be uniquely determined by the identifiers and the types of the components. Alphabetic order is perhaps the most natural choice.

4.1 Optimization of standard formals

Consider compilation of a formal *atomf*[*x*, *t*] where the values of *x* and *t* are known at compile time. If the result of clause *e* is known to be of type *t*, invocation (*atomf*[*x*, *t*]*e*) is equivalent to the clause **env**(*x* = *e*). The memory representation of the environment is identical with the representation of the value of *e*. Thus the invocation can be compiled simply into *code*(*e*). Procedure *atomf* generates no code at all.

Next consider a call

$tuplef[f_1, \ldots, f_n][e_1, \ldots, e_n]$

By the definition of *tuplef* and by associativity of *econcat*, the call is equivalent to

$econcat[(f_1 e_1), \ldots, (f_n e_n)]$

Let *E* denote the result of the call. Assume that each $(f_i\ e_i)$ returns an environment with exactly one bound identifier x_i, which is known at compile time, and assume that identifiers x_i are distinct. Components of *E* can then be stored in consecutive memory locations as noted above. Let $i_1, \ldots, i_n$ be the order of the components in the memory. The call can be compiled into the following code:

$code(e_{i1})$
$code(f_{i1})$
$code(e_{i2})$
$code(f_{i2})$
...
$code(e_{i_n})$
$code(f_{i_n})$

If some formals f_i yield environments with more than one binding, it may be necessary to permute the components of the environment after evaluating all invocations $(f_i\, e_i)$. For example, if f_1 returns an environment with identifiers "a" and "c" and if f_2 returns an environment with identifiers "b" and "d", the call

$tuplef[f_1, f_2][e_1, e_2]$

can be compiled into the following code (assuming that the components of environments are stored in an alphabetical order):

$code(e_1)$
$code(f_1)$
$code(e_2)$
$code(f_2)$
code for swapping the "b" and "c" components

Data transfer instructions can often be reduced by leaving empty holes in the stack when the values of component formals are evaluated.

4.2 Optimization by symbolic evaluation

More complicated formals can often be reduced into simpler ones by symbolic evaluation. The resulting program may then be susceptible to the optimization techniques discussed above. From the semantics of the language, the following evaluation rules can be derived:

1. By definition, clause (**proc** $e_1 \Rightarrow e_2)e_3$ can be reduced to **use** $e_1\ e_3$ **in** e_2. Invocations of standard procedures can be evaluated using their defining equations.
2. Clause (**if** *true* **then** e_1 **else** e_2) reduces to e_1 and (**if** *false* **then** e_1 **else** e_2) reduces to e_2.
3. Clause (**case** e **in** $f_1 \Rightarrow e_1, \ldots, f_n \Rightarrow e_n$ **else** $f_{n+1} \Rightarrow e_{n+1}$) reduces to (**use** $f_i\ e$ **in** e_i), where f_i is the first such formal that $(f_i\, e)$ does not fail. If all invocations $(f_i\, e)$ fail, the case-clause reduces to (e; **abort**). In the latter case the clause e can be eliminated if the compiler can conclude that e has no side effect. Note that the actual value of the clause e need not necessarily be known.
4. Clause (**use** **env**$(x_1 = e_1, \ldots, x_n = e_n)$ **in** e) can, under certain conditions, be reduced by substituting the occurrences of x_i with e_i in e; the substituted e replaces the use-clause. This reduction rule can always be applied if clauses e_i have no side effects. But even if e_i does have a side effect, the substitution is legal if x_i *occurs in e exactly once*. The freedom in the evaluation order of the components of a use-clause makes the rule both simpler and more general. If left to right evaluation were guaranteed in use-clauses, an additional constraint would be required: identifier x_i can be replaced by e_i in e only if there is no subclause in e that precedes the occurrence of x_i and may have a side effect. This additional constraint is actually satisfied in most cases that occur in practice, but the compiler may have difficulty in verifying it.
This rule, in combination with rule 1, may lead to a nonterminating sequence of reductions. Application of the rules should therefore be restricted by suitable heuristics.
5. The first component of a serial clause $(e_1; e_2)$ can be moved into the front of a structured clause in the following cases:

[. . ., $(e_1; e_2)$, . . .]
proc$(e_1;\ e_2) \Rightarrow e_3$
$(e_1;\ e_2)\ e_3$
$e_3(e_1;\ e_2)$
env(. . .$(e_1;\ e_2) = e_3$, . . .)
use$(e_1;\ e_2)$ **in** e_3
if $(e_1;\ e_2)$ **then** e_3 **else** e_4
case $(e_1;\ e_2)$ **in** $e_{11} \Rightarrow e_{12}, \ldots$

4.3 Formals as macros

Implementation of the language defined in section 2 is complicated since the compiler must be prepared for runtime type checks and dynamic binding even though they may be seldom actually used. The implementation must also provide an alternate representation for dynamic environments and an exception mechanism for abort-clauses. Dynamic binding and runtime type checks may also be incompatible with other desirable features in the design of new languages.

These problems can be avoided if formals are treated as macros that must be evaluated completely at compile time. Different keywords can be used to distinguish compile-time clauses from runtime clauses. For instance, compile-time clauses could begin with keywords **#if**, **#proc**, **#case** etc. Macros are evaluated using the rules discussed in the preceding section. Since some features of the language are of little use at compile time and others at runtime, semantic restrictions can be added to facilitate implementation. The list of restrictions might include the following:

- The identifiers bound by environments must be known at compile time. This restriction implies static binding.
- Type checks must be evaluated at compile time except when a runtime type check is explicitly requested with a case-clause.
- Failures are not propagated at runtime. In other words, execution of an abort-clause at runtime is a fatal error.
- Standard procedures *atomf*, *nullf* and *fconcat* are available only at compile time.
- Variable assignments are available only at runtime.

There are many additional details to be filled – for instance, there are several possible ways to define the type inference rules. Discussion of these decisions would require much space and is beyond the scope of this article. However, there is one potential problem that deserves a note, namely the termination of macro evaluation, which is undecidable in the general case. An additional restriction is therefore necessary if termination of compilation is to be guaranteed. Analysis of example formals reveals that recursive formals tend to have tuples as arguments and either the length of the tuple or the length of one of its components usually decreases at each recursion level. Termination of macros can be guaranteed by requiring that a macro can be called recursively only if the size of the argument decreases on each recursion level, size being defined by

$$size([x_1, \ldots, x_n]) = n + 1 + \sum_{i=1}^{n} size(x_i)$$

$$size(x) = 1 \quad \text{if } x \text{ is not a tuple}$$

5. CONCLUSIONS

Most parameter transmission mechanisms, implicit conversions, pattern-based parameter binding and optional, repeatable and variable-type parameters can be defined as special cases of an abstract parameter transmission mechanisms, where the formal parameter part of a procedure is a mapping from the set of arguments into the set of environments. Abstract formals can be implemented efficiently in most cases that are of practical interest.

Acknowledgements

The author wants to thank Prof. Reino Kurki-Suonio and Prof. Ralph-Johan Back for their helpful comments.

REFERENCES

1. *Specification for Computer Programming Language Pascal*. International Organization for Standardization, Switzerland (1983).
2. B. Stroustrup, *C++ Programming Language*. Addison-Wesley (1986).
3. H. Stoyan, *Lisp-programmierhandbuch*. Akademie-Verlag, Berlin (1978).
4. V. R. Prasad, Variable number of parameters in typed languages. *Software – Practice & Experience* **10**, 507–517 (1980).
5. G. Ford and B. Hansche, Optional, repeatable and varying type parameters. *SIGPLAN Notices* **17:2**, 41–48 (1982).
6. O.-J. Dahl, B. Myrhaug and K. Nygaard, *Common Base Language*. Norwegian Computing Centre (1970).
7. P. Wegner, On the unification of data and program abstraction in Ada. *Proceedings of the 10th conference on Principles of Programming Languages* 257–264 (1983).
8. D. Gelenter, S. Jagannathan and T. London, Environments as first class objects. *Proceedings of the 14th conference on Principles of Programming Languages* 98–110 (1987).
9. R. Burstall and B. W. Lampson, A kernel language for abstract data types and modules. *Proceedings of the International Symposium on Semantics of Data Types*. Sophia-Antipolis, France, 1–50 (1984).
10. A. van Wijngaarden *et al.*, *Revised Report on the Algorithmic Language Algol 68*. Springer-Verlag (1976).

Announcements

10–13 SEPTEMBER 1990

SWISS FEDERAL INSTITUTE OF TECHNOLOGY, ZÜRICH

Joint Conference on Vector and Parallel Processing

CONPAR 90
VAPP IV

The past decade has seen the emergence of the two highly successful series of CONPAR and of VAPP conferences on the subject of parallel processing. The *Vector and Parallel Processors in Computational Science* meetings were held in Chester (VAPP I, 1981), Oxford (VAPP II, 1984) and Liverpool (VAPP III, 1987). The *International Conferences on Parallel Processing* took place in Erlangen (CONPAR 81), Aachen (CONPAR 86) and Manchester (CONPAR 88). The format of the joint meeting will follow the pattern set by its predecessors. It is intended to review hardware and architecture developments together with languages and software tools for supporting parallel processing. Another objective of the conference will be to highlight advances in algorithms and applications software on vector and parallel architectures.

It is expected that the programme will cover:

- languages/software tools
- hardware/architecture
- algorithms/software
- applications

Also special sessions will be devoted to the field of application and/or programming language specific architectures; i.e. machines, where performance has been gained through limiting the field of applications, or systems designed according to a joint optimization of programming language and architecture.

Other topics of special interest are:

- performance analysis for real-life applications
- testing and debugging of parallel systems
- portability of parallel programs
- paradigms for concurrency and their implementation

The conference should appeal to anyone with an interest in the design and use of vector and parallel machines.

Exhibition and Posters

An exhibition will be organized of:

- vendor products
- current research work and noncommercial system prototypes

related to the conference theme.
Poster sessions will be organized.

For further information contact:

Prof. Dr. Helmar Burkhart, Institut für Informatik, Universität Basel, Mittlere Strasse 142, CH-4056 Basel, Switzerland. Tel: +41 61 449967. e-mail: burkhart urz.unibas.ch

6–8 JUNE 1990

Eurographics Workshop on Object-Oriented Graphics, Königswinter, Federal Republic of Germany

Aims and scope

Object-oriented methods are proving to be particularly applicable to computer graphics – in formulating new graphics standards and in dynamic graphics and human-computer interaction. Specific computer graphics problems have also resulted in a critique of the object-oriented paradigm.

Venue and fee

The workshop will be held at Königswinter near Bonn, on the Rhine. The fee will be around DM 700, including accommodation, meals, evening boat excursion and reception at Birlinghoven Castle. Student participants without access to other funds could be subsidised.

Co-chairmen

Peter Wißkirchen (GMD) and Edwin Blake (CWI).

Information

Requests for information should be sent to the workshop secretary:
Ms. Marja Hegt, O-O Graphics Workshop, CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands. Tel: +31 20 592 4058. Fax: +31 20 592 4199. Email: marja@cwi.nl (uucp).