Uniform approach to parameter transmission mechanisms, coercions, optional parameters and patterns*

MATTI JOKINEN

Computer Center
University of Turku
SF—20500 Turku
Finland

Abstract

The formal parameter part of a procedure can be regarded as a mapping from the set of arguments into the set of environments. If environments and environment-valued functions are treated as first-class objects, a number of useful linguistic features can be constructed from a small set of elementary building blocks; such features include the most parameter transmission mechanisms, implicit conversions, conditional clauses based on pattern matching, and optional, repeatable and variable-type parameters.

1. Introduction

Programming languages use numerous variants of mappings of the general form $S \rightarrow V$, where S is a finite set of character strings and V is the universe of data objects. Such mappings can be divided into three main categories:

- *Evaluation environments* bind the free identifiers of programs into data objects. Although they are defined by declarations embedded in the program text, they tend to belong to the meta universe outside the domain of data objects. Most programming languages provide no method of identifying them by name or referring to them as entities.
- *Packages* are used as library modules, and their components are mostly types and procedures. They are often used for information hiding. They are typical second-class objects which may have names but must be completely defined at compile time.
- *Records* are designed for storing runtime data. In most modern programming languages they are first-class objects which can be created and modified at runtime.

The distinction between the three concepts makes implementation simpler, but conceptually it is more or less arbitrary. Advantages of a uniform approach are obvious [1, 4, 9]. The idea of combining the concepts is not new: Simula classes [2] are used in all three roles.

* Lecture presented at the 1st Finnish—Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

2. Language

Records, packages and evaluation environments are treated uniformly in this paper and they are all called environments. We shall design a programming language that uses environments extensively as first-class objects. Details of syntax and semantics are of minor interest here, and the language will not be defined rigorously; it is solely a tool for discussing various cases where environment-valued functions prove to be useful.

2.1. Environments

An environment can be created with a clause

$$\{e_{11} \mapsto e_{12}, \ldots, e_{n1} \mapsto e_{n2}\}$$

where e_{ij}s are arbitrary expressions. Clauses e_1 and e_2 can be evaluated in any order or interleaved; this allows some extra freedom in optimization. Each e_{i1} must evaluate to a string. The resulting environment binds the strings to the values of expressions e_{i2}. 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. An environment can also be used in a clause

$$\textbf{with } e_1 \textbf{ do } e_2$$

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. The whole program is implicitly embedded in an environment that contains the definitions of standard identifiers.

2.2. Procedures

The definition of a procedure usually looks something like

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

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. In the simplest case 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. But in many programming languages parameter transmission is more complicated. 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 values of several formals, as conformant array parameters in Pascal [6]. Implicit

actions allow a more compact notation and their proper use may thereby improve readability. Unfortunately the rules are usually built into the language, and although modern languages allow the definition of application-specific types they rarely [8] provide any way to extend implicit actions to user-defined types.

The FEXPR feature of Lisp [7] 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 Ford and Hansche [3] and Prasad [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.

If parameters are passed by value, the call is equivalent to the following with-clause:

$$\textbf{with } \{"x_1" \mapsto e_1, \ldots, "x_n" \mapsto e_n\} \textbf{ do } e.$$

Thus the formal parameter part $(x_1\colon t_1, \ldots, x_n\colon t_n)$ can be regarded as a function that maps the argument tuple into an environment. Since environments are first-class objects, it is natural to 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. A procedure object is created with a clause

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

where e_1 is an arbitrary clause that evaluates to an environment-valued procedure (from now on all such procedures will be called formals).

It is convenient to reduce multi-argument procedures into single-argument procedures by treating the argument list as a tuple. A tuple object is created with a clause $[e_1, \ldots, e_n]$. A one-element tuple is *not* identical with its element. Expressions e_i can be evaluated in an arbitrary order, or interleaved. Procedures with no parameters 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 **proc** $f \Rightarrow b$, the invocation is equivalent to **with** $f e_2$ **do** b. For convenience, certain operators will be written in their familiar infix or postfix notation. For example, 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.

$$atomf[s,t]\,x = \begin{cases} \{s \mapsto x\}, & \text{if } x \text{ is of type } t. \\ \textbf{abort} & \text{otherwise} \end{cases}$$

If x is not of type t, the call causes a failure, a termination without any result, represented by the clause **abort**. Failures can be trapped in case-clauses, as will be seen later; untrapped failures are propagated to upper-level clauses. 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,

$$atomf["n", int]\,4 = \{"n" \mapsto 4\}.$$

To make formals look more familiar, the following sugared syntax is defined for calls of *atomf*:

$$x{:}t = atomf[x, t].$$

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

$$\begin{aligned} &nullf[\,] = \{\,\} \\ &nullf\,x \;= \textbf{abort}, \quad \text{if } x \neq [\,]. \end{aligned}$$

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.

$$\begin{aligned} &fconcat[f_1,f_2][e_1, \ldots, e_n] = econcat\,[(f_1\,e_1), f_2[e_2, \ldots, e_n]] \\ &fconcat[f_1,f_2][\,] = \textbf{abort} \\ &fconcat[f_1,f_2]\,x = \textbf{abort}, \quad \text{if } x \text{ is not a tuple.} \end{aligned}$$

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

$$\begin{aligned} &tuplef[\,] = nullf \\ &tuplef[f_1, f_2, \ldots, f_n] = fconcat\,[f_1, tuplef[f_2, \ldots, f_n]]. \end{aligned}$$

2.4. Types

Since type checks occur at runtime, there must be a sensible action taken when a type check fails. A failing type check is defined 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 *anyenv* 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

Many modern languages have union types and a conditional clause that allows a safe access to the contents of a union. Such a clause will be needed in all the examples below. The syntax and semantics of the clause can be defined elegantly with generalized formals. The syntax 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, 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, the next formal is tried. If all the formals f_1 to f_n fail, 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, or if there is no else-part, the case-clause fails.

3. Applications

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

Here the case-clause is used to compute the argument of (*id*: *real*). Type **union**[*int*,*real*] could be replaced with the type *any*. Formal *intreal*[*x*] would normally be used in definitions of arithmetic functions. However, *atomf*[*x*, *real*] could be used 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 *eps*. The header of the procedure might look like this:

```
proc tuplef ["f": real→real,
             intreal "a",
             intreal "b",
             "eps": real]⇒ ...
```

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

3.2. Parameter 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. *Rcp_value* generates procedures that compute values of recipes:

```
rcp_value=proc ("t": type)⇒
    proc ("x": ref union[t, void→t])⇒
        case x↑ in
            ("y": t)⇒y,
            ("f": void→t)⇒(with {z⇝f[ ]} do (x:=z; z)).
```

Here $x\uparrow$ denotes the contents of the cell pointed to by x. Components of the serial clause $(x:=z;\ z)$ are evaluated from left to right, and the value of the clause is the value of the last component. The other auxiliary procedure *rcpdefs* just generates two shorthand notations, *rcp* and *u*:

```
rcpdefs=proc ("t": type)⇒
        {"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["id": string, "t"; type]⇒
    with rcpdefs t do
        proc("x": u)⇒
```

```
                {id↦case x in  ("y": t)⇒y,
                               ("p": ref t)⇒p↑,
                               ("f": void→t)⇒f[ ],
                               ("r": rcp)⇒rcp_value t r}

name=proc tuplef["id": string, "t": type]⇒
    with rcpdefs t do
        proc("x": u)⇒
            {id↦case x in  ("y": t)⇒(proc nullf⇒y),
                           ("p": ref t)⇒(proc nullf⇒p↑),
                           ("f": void→t)⇒f,
                           ("r": rcp)⇒(proc nullf⇒rcp_value t r)}

need=proc tuplef["id": string, "t": type]⇒
    with rcpdefs t do
        proc("x": u)⇒
            {id↦case x in  ("y": t)⇒new rcp y,
                           ("p": ref t)⇒new rcp (p↑),
                           ("f": void→t)⇒new rcp f,
                           ("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["id": string, "t": type]⇒
    with rcpdefs t do
        proc("x": u)⇒
            {id↦case x in  ("y": t)⇒new t y,
                           ("p": ref t)⇒p,
                           ("f": void→t)⇒new t (f[ ]),
                           ("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.

3.3. Procedures with varying number of parameters

Procedures with optional parameters 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["head": t, "tail": anytuple]
```

Another possibility is to define optional arguments that receive 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 i^{th} element corresponds to the i^{th} element of the tuple L. The length of A may be smaller than the length of L, in which case the missing elements are given default values from L.

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

where

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

If there are many optional parameters, it is more convenient to identify them by name than by position. In the list of actual arguments, an optional argument is specified 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 ("L": anytuple)⇒
    proc ("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 ("L": anytuple)⇒
    case L in
        nullf⇒{},
        fconcat[tuplef["name": string, "t": type, "default": any],
                "tail": anytuple]⇒
            econcat[(name: type) t, types tail]

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

3.4. Patterns

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 ("p"): any)⇒
    case p in
        nullf⇒nullf,
        ("s": string)⇒if s[1]≧'A' and s[1]≦'Z'
                      then (s: any)
                      else (proc("t": string)⇒
                                  if s=t then {} else abort),
        fconcat["head": any, "tail": anytuple]⇒
                      fconcat [pattern head, pattern tail]
```

For example, the value of the clause

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

is a formal that accepts 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 (for example, strings beginning with a capital letter). These features can be defined in the language without difficulty.

4. Implementation

The programming language designed in the preceding sections is based on late binding and runtime type checks. That is typical of interpreted languages, and the reader may wonder whether the ideas presented in this paper are of any use in compiled languages where efficiency is considered more important. Fortunately the quality of the code can be greatly improved with relatively simple optimization methods.

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 in which 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 of the base address of the environment. A single-element environment $\{x \rightsquigarrow v\}$ is represented exactly as the object v. Assume that in an invocation $(p\ e)$ the value of p is completely known at compile time and defined by

$$p = (\mathbf{proc}\ (x\colon t) \Rightarrow u)$$

If, in addition, x is a string constant and e is guaranteed to be of type t, the environment produced by the formal can be used as the lower part of the activation record of a procedure as in conventional languages and the invocation can be translated into the instruction sequence

$$code(e);\ \text{jsub}(u)$$

where $code(e)$ evaluates e and leaves its value on the top of the stack, and jsub(u) saves the program counter and transfers control to the body u of the procedure.

Next assume that p is defined by

$$p = (\textbf{proc}\ tuplef[f_1, \ldots, f_n] \Rightarrow u)$$

where each f_i is completely known at compile time, e_i is known to be of type suitable as an argument for f_i, and the result of f_i is a mini-environment $\{x_i \rightsquigarrow v_i\}$ where x_is are string constants and $x_i \neq x_j$ whenever $i \neq j$. The invocation can now be translated into the instruction sequence

$$code(f_1\ e_1);\ \ldots;\ code(f_n\ e_n);\ \text{jsub}(u).$$

Procedure calls involving more complicated formals can usually be optimized with partial evaluation. From the semantics of the language the following evaluation rules can be derived:

1. Clause (**proc** $e_1 \Rightarrow e_2) e_3$ can, by definition, be reduced to (**with** e_1 e_3 **do** e_2).
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 (**with** f_i e **do** 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 be known.
4. Clause (**with** $\{x_1 \rightsquigarrow e_1, \ldots, x_n \rightsquigarrow e_n\}$ **do** e) can, under certain conditions, be reduced by substituting the occurrences of x_i with e_i in e; the substituted e replaces the with-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.* If left to right evaluation is to be guaranteed, an additional constraint is 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 difficulties in verifying it. The rule becomes simpler and more general if the requirement of left-to-right evaluation is relaxed.
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:

   $[\ldots, (e_1; e_2), \ldots]$
   proc $(e_1; e_2) \Rightarrow e_3$
   $(e_1; e_2) e_3$
   $e_3\ (e_1; e_2)$
   $\{\ldots (e_1; e_2) \rightsquigarrow e_3, \ldots\}$
   $\{\ldots e_3 \rightsquigarrow (e_1; e_2), \ldots\}$

with $(e_1; e_2)$ **do** e_3
if $(e_1; e_2)$ **then** e_3 **else** e_4
case $(e_1; e_2)$ **in** $e_{11} \Rightarrow e_{12}, \ldots$

The reader is encouraged to apply the rules to the formals defined in the preceding section.

Rules 1 and 4 together may lead to a nonterminating sequence of reductions. Since compilers have difficulties in recognizing the diverging clauses, it is probably better to let the programmer specify which clauses shall be evaluated at compile time. Abstract formals could then be regarded as sophisticated macros rather than ordinary procedures.

Acknowledgments

The author wants to thank Reino Kurki-Suonio and Robert Johnson for their helpful comments.

References

[1] BURSTALL R. and LAMPSON B. W., '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).
[2] DAHL O-J., MYRHAUG B. and NYGAARD K., *Common Base Language*, Norwegian Computing Centre (1970).
[3] FORD G. and HANSCHE B., 'Optional, repeatable and varying type parameters', *SIGPLAN Notices* **17:2**, 41—48 (1982).
[4] GELENTER D., JAGANNATHAN S. and LONDON T., 'Environments as first class objects', *Proceedings of the 14th conference on Principles of Programming Languages*, Munich, West Germany, 98—110 (1987).
[5] PRASAD V. R., 'Variable number of parameters in typed languages', *Software—Pratice & Experience*, **10**, 507—517 (1980).
[6] *Specification for Computer Programming Language Pascal*, International Organization for Standardization, Switzerland (1983).
[7] STOYAN H., *Lisp-programmierhandbuch*, Akademie-Verlag, Berlin (1978).
[8] STROUSTRUP B., *C++ Programming Language*, Addison-Wesley (1986).
[9] WEGNER P., 'On the unification of data and program abstraction in Ada', *Proceedings of the 10th conference on Principles of Programming Languages*, Austin, Texas, 257—264 (1983).
[10] VAN WIJNGAARDEN A. et al., *Revised Report on the Algorithmic Language Algol 68*, Springer-Verlag (1976).