STUDIA UNIV. BABEŞ-BOLYAI, INFORMATICA, VOLUME II, NUMBER 1, 1997

MODELLING AND IMPLEMENTING PARAMETER PASSING METHODS

ALEXANDRU VANCEA MONICA VANCEA

Abstract. The paper presents a parameter passing techniques specification method, a model which helps their correct and efficient implementation. By using these abstactions at the im- plementation level we could exactly specify the transmission mechanisms and we could approach the problems posed by some particular actions that are necessary when employing these methods.

1. Preliminaries

The issue of parameter transmission is of a great importance for procedure calls implementation. This is not aimed only to obtain more efficient versions of procedure calls, but also to generate correct procedure calls implementations, which, unfortunately, still depends upon the details of transmission of one or another particular parameter passing method.

In this matter is important to notice the trend brought by Ada [1], which succeeds to make abstraction of these details at the programmer's level. In Ada we have three kinds of parameters:

- **in** - semantics equivalent to call by value, the flow of data is from the caller's environment to the procedure and the value of actuals cannot be changed;
- **out** - semantics equivalent to call by result, the flow of data is from inside the procedure to the outer environment and the role of such actuals is only to communicate results of computations;
- **in-out** - semantics equivalent to call by reference, the actuals enter the procedure with some needed values which can be changed inside and then "transported" outside.

Even if we mentioned this classification in relation with the semantics of some well-known parameter passing methods, let's notice that we are only interested in their semantics and we do not discuss particular ways in which they could be accomplished.

Anyway, when developing code in many programming languages (as Pascal or C for example) we must consider these passing details at the programmer's level.

In order to develop the expressive power of a programming language, one has to provide means to express as accurately as possible the implementation of its features. In [2] and in many related papers we find descriptions of procedure calls implementations. One of the aspects not detailed there is how can we describe (and most important how

Received by the editors: July 6, 1997.
1991 Mathematics Subject Classification. 68N15, 68Q60.
1991 CR Categories and Descriptors. D.3.3 [**Programming Languages**]: Language Constructs and Features *procedures, functions* and *subroutines*; *data types* and *structures;* D.3.1 [**Programming Languages**]: Formal Definitions and Theory – *semantics, syntax.*

can a compiler take further advantage of a good metalanguage description) the implementation of a parameter passing method.

For this purpose we present here **PARTRAN** (our adapted version of the metalanguage presented in [4]) which is applied further in the description of the most wided used parameter passing methods. We can use further these descriptions in the context described in [2] which corresponds to the correct completion of the PAR field of the activation record that is built for any particular procedure call.

2. The PARTRAN metalanguage abstractions

In this section we describe the abstractions of a metalanguage (which we called **PARTRAN**) which are adapted from the results presented by Jokinen in [4].

A. A *procedure* can be defined and represented as

$$\textbf{procedure}\ pname(x_1{:}t_1,\ldots,x_n{:}t_n)\{body\}$$

and a procedure call as

$$pname(a_1,\ldots,a_n)$$

where a_i is of type t_i.

B. One of the most important elements introduced in our language is the use of the *environment* concept. We will consider in **PARTRAN** that each procedure activation has a manifest association with its environment (that is the environment in which its activation takes place). That's why we will use for our procedures the term *environment-valued procedures*.

Why do we need them? The motivation resides in the fact that we cannot make abstraction that for all significant actions (namely evaluations) that take place in the parameter transmission phase, we need also the precise environment(s). For example, the semantics of the *call_by_value* parameter passing method makes the body of *pname* to be evaluated in an environment in which each x_i is bound to the value of a_i

$$By \qquad env(id_1 = e_1,\ \ldots,\ id_m = e_m)$$

we denote the environment in which an expression using some of the specified identifiers will be evaluated. Practically, an environment is a mapping from a finite set of strings into data objects [3]. The result of this mapping are the bindings made between identifiers id_i and the values of expressions e_i.

An environment can be used in a clause

$$eval\ exp\ using\ env$$

where *env* evaluates to an environment and *exp* is an expression which value will be obtained through evaluation in the environment yielded by *env*. The value of this clause is the value of *exp* whose free identifiers are bound as in the environment produced by *env*.

C. A *procedure object* is created with a clause

$$proc\ e_1 : e_2$$

where e_1 is an expression that evaluates to an *environment-valued procedure* (EVP) and e_2 is the body of the procedure.

D. Another interesting idea is the use of *higher order functions* as generators for some other specific regular functions needed in our descriptions. We will introduce such a higher order procedure, named *parform*, which generates an EVP. It accepts as an argument a pair *[s,t]*, where *s* is a *string* and *t* is a *type*. The value of the invocation

$$parform[s,t]$$

is an (environment-valued) 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:

$$parform[s,t]x = \begin{cases} env(s = x), if\ \ x\ is\ of\ type\ t \\ fail \qquad otherwise \end{cases}$$

s is an arbitrary string-valued expression and it is the value of *s* (rather than the identifier *s*) that becomes bound in the environment. The motivation of introducing higher order functions is for generating functions and procedures which will act as agents for accomplishing the required actions at the moment of parameter transmission.

E. The *case-clause* is introduced in the syntax

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

where the values of clauses f_1 **to** f_{n+1} are EVPs. The **else**-part is optional. The clause is evaluated by first evaluating the expression *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 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.

Regarding the types involved, we will assume the use of standard types *int*, *real*, *string* and *type* (meaning any type) and the type constructors **ref**, **union**, **tuple** and **-->**. Type **ref** *t* is the type of pointers to *t*-typed cells. **union** $[t_1,...,t_n]$ represents the union of the specified types $t_1,...,t_n$. Type **tuple** $[t_1,...,t_n]$ is the type of tuples $[x_1,...,x_n]$ where x_i is of type t_i. The type of functions with domain t and range u is denoted by *t --> u*.

Example. Let's take a simple example: we want to define a generator for EVPs that accept either a digits-string or an integer as their actual argument and convert it into the corresponding integer value in the former case and let it unchanged in the latter case (the transformation is actually done by the function *stringtoval* - the equivalent of the Turbo Pascal VAL function):

```
stringval = proc parform["id", string] :
                proc parform["x", union[string, int]] :
                    proc parform[id, int] :
                     (case x in
                          parform["i", int] : i,
                          parform["s", string] : stringtoval s)
```

3. Metalanguage at work

The **PARTRAN** metalanguage promotes the idea that parameter transmission mechanisms are closely related to types. So, transmissions by different mechanisms can be reduced to passing various types of data.

Call by value means just transmitting the required value of the type involved in the transmission. *Call by reference* is equivalent to the transmission of a parameter of type **ref** *t*. *Call by name* is equivalent to the transmission of a parameter of type *void --> t*, where void can be defined by an empty tuple.

We first define two procedures which will help us further. *Rep_value* generates procedures that compute values of an object called *recipe*, which is an object of type **ref union** *[t,void--> t]*.

```
rep_value = proc parform["t", type] :
                    proc parform["x", ref union[t, void --> t]] :
                      case x^ in
                           parform["y", t] : y,
                           parform["f", void --> t] : (x:=f[]; x^)
```

*x^* denotes the contents of the cell pointed by **x**. The other procedure, ***rcpdefs***, just generates two shorthand notations, **rcp** and **u**. By *new t y* we denote the action of allocation of a new cell of type t and the returning a pointer to that cell in y.

```
rcpdefs = proc parform["t", type] :
                env("rcp" = ref union [t, void --> t]
                    "u"   = union [t, void --> t , ref t, rcp])
```

Following this definition, *rcp* represents the type union of the type denoted by t with the type of all functions without parameters which return t type values. The idea of defining functions without parameters to be used in expressions or parameters evaluation is due to Ingerman [1] which promoted the concept of *thunk*, used in the implementation of *call_by_name.*

Call by *value*, *name* and *reference* can now be defined with the following procedures.

```
value = proc (parform["id", string], parform["t", type]) :
            eval {
              proc parform["x", u] :
                 env(id =
                      case x in
                        parform["y", t] : y,
                        parform["p", ref t] : p^,
                        parform["f", void --> t] : f[],
                        parform["r", rcp] : rep_value t r)
                                  } using rcpdefs t;
```

In this *call_by_value* description, *id* represents the formal parameter (more precisely, *parform["id", string], parform["t", type*] declares that *id* and *t* will be identifiers of the specified types in the description of the procedure body) and *x* represents the actual parameter. Upon the type value of *u*, the case clause will choose the proper actions (namely the correct evaluations and the quantities to be transmitted) to be performed in the CALL and ENTRY phase (see [1] for details).

Let's notice that *value* is obtained as the application of a higher order function. This higher order function describes the body of the EVP which has to be applied in the process of parameter transmission. The same observation holds for the procedures *name* and *reference*.

```
name = proc (parform["id", string], parform["t", type]) :
          eval {
           proc parform["x", u] :
            env(id =
                case x in
                      parform["y", t] : (proc env():y),
                      parform["p", ref t] : (proc env():p^),
                      parform["f", void --> t] : f,
                      parform["r", rcp] :
                      proc env(): rep_value t r))} using rcpdefs t

reference = proc (parform["id", string], parform["t", type]) :
              eval {
                proc parform["x", u] :
                   env(id =
                        case x in
                          parform["y", t] : new rcp y,
                          parform["p", ref t] : new rcp (p^),
                          parform["f", void --> t] : new rcp f,
                             parform["r", rcp] : r)
                         } using rcpdefs t
```

Such abstractions can be useful in expressing and solving all the situations in which we can be at the moment of the parameters transmission. The completion of the PAR stage [2] means essentially the action of the appropriate procedure, namely one from above, depending on the actually choosed method.

4. Conclusions

One of the most problematic situation that appears in the parameters transmission process is the case when we have functional parameters as formals. Solutions to the implementation of such transmissions can be found in [2]. The advantage of PARTRAN is that it could model such cases by means of higher order functions which generate and describe the behaviour of functional parameters transmission.

It would be interesting in the next intended step to include these facilities into an experimental compiler for a subset of a programming language, to be able to evaluate the practical results at which we could aim [5]. These are mainly two:

- an efficient implementation of the required parameter passing technique;

- the possibility offered to the programmer to interfere with the compiler and to specify some needed actions based on the platform offered by the modelling language presented here;

The practical effects are still to study.

REFERENCES

[1] **Horowitz E.**, *Fundamentals of Programming Languages*, Springer Verlag, 1983.

[2] **MacLennan B.**, *Principles of Programming Languages. Design, Evaluation and Implementation*, CBS College Publishing, 1983.

[3] **Gelernter D. et al.**, *Environments as first class objects*, in Proceedings of the 14th Conference on Principles of Programming Languages, pp.98-110, 1987.

[4] **Jokinen M.**, *Parameter Transmission Abstractions*, in The Computer Journal, vol.33, no.2, 1990, pp.133-139.

[5] **Tai Kuo-Chung**, *Comments on Parameter Passing Techniques in Programming Languages*, ACM Sigplan Notices, 17, 2, 1982.

Babeş-Bolyai University, Faculty of Mathematics and Informatics, RO 3400 Cluj-Napoca, str. Kogalniceanu 1, România.

E-mail address: `vancea@cs.ubbcluj.ro`

Babeş-Bolyai University, Faculty of Economics, RO 3400 Cluj-Napoca, str. Kogalniceanu 1, România.

E-mail address: `vancea@econ.ubbcluj.ro`