A paper submitted to SIGPLAN Notices 15-Apr-1989

XE Design Rationale: Clu Revisited

Vesa Hirvisalo
Jari Arkko
Juha Kuusela
Esko Nuutila
Markku Tamminen

Helsinki University of Technology
Department of Computer Science
Otakaari 1, 02150 Espoo 15, FINLAND

Abstract

XE is a programming language designed and implemented at the Helsinki University of Technology by the ExBed project. The design of XE is based on the CLU programming language. This paper discusses differences between the two languages and the design decisions of XE.

1. Background

The XE language [Arkk89] and its programming environment have been developed within the ExBed project, which was established to develop software tools for construction of embedded expert systems. The project needed a general purpose programming language with added facilities for rule-based programming. High level abstraction mechanisms and the possibility for efficient static semantic analysis were considered crucial for the language. The main reason for this was the wish to perform as much verification and optimization as possible at compile time to satisfy the stringent requirements of embedded systems.

There existed no programming language that satisfied our needs and therefore we decided to design a new programming language. Our first result, XC [Nuut87], was based on C++ [Stro86], and it was implemented using preprosessor techniques. We found out that the C++ approach had several major drawbacks [Arkk88]. Also the preprocessor techniques could not give enough support for building a satisfactory programming environment.

The CLU language, designed and implemented at MIT [Lisk81], was close to our needs. So, we selected CLU as a basis for the design of XE. Because of the licence policies of CLU and our need for an integrated software development environment we decided to write a proper compiler for the language, not a preprocessor for CLU. We found out that the built-in library of CLU did not fit our needs and that the rule-based expression mechanism of our choice required very basic new facilities. Therefore XE was not designed as a direct extension of CLU.

The structure of this paper is as follows. Chapter 2 describes the rule abstraction of XE and some language constructs closely related to it. In chapter 3 we discuss the parameterization (generic modules). The type system and built-in types are described in chapter 4. There exists quite a number of other differences between CLU and XE. Some of these features are discussed in chapter 5. The reader is assumed to be familiar with the CLU language.

2. Rules

To support rule based programming a rule abstraction was included in XE. A rule is given zero or more argument objects and it creates a sequence of items, that it yields one at a time. Thus the interface of an XE rule resembles a CLU iterator. The items yielded usually contain data that satisfies the condition of the rule. The condition can be quantified. The following is an example of a simple XE rule (from a file system expert):

```
find_movable_file = rule(root_directory: string) yields(string, proctype(string))
  when [for directory: string in directories(root_directory)]
          [for file_name: string in files(directory)]
           file_name_suffix(file_name) = "tcr":
    yield(file_name,
           proc(new_name: string) binds(file_name)
              move_file(file_name, new_name)
           end)
end
```

The rule searches files with the suffix *trc* from a given directory tree. Their names and the actions to move them to different places are yielded to the invoker of the rule. The invoker (typically called inference engine in rule-based programming) can then decide which actions to execute.

Rules often return actions that must be executed later. To enable the coding of actions as routines defined by a rule the concept of nested modules was included in XE. With nested modules it is also possible to write special purpose auxiliary routines that are often needed when rule sets or other collections of routines are written. In CLU this kind of routines must be made global.

To make the use of rules easier, the use of nameless routines and lexical closures were allowed in XE. An XE routine can bind variables declared in its lexical environment. A routine can be passed as an object and these bindings remain in effect even if the routine is used elsewhere than in its own lexical environment. There exist no global variables in XE. Therefore the external variable bindings must be explicitly declared in the routine header. The binding mechanism is also useful in allowing more flexibility in the typing of rules. If all bound variables were required to be passed as arguments it would be difficult to obtain uniformly typed rulesets.

3. Parameterization

The parameterization abstraction of CLU was found to be too restricted. In CLU it is not possible to write type generators such as `record` even though this kind of generators are included in the built-in library of the language. In XE the user can write *selector generators* that are parameterized by a field list. A field list is a list of name-type pairs, such as in `record{name: string, age: int}`.

The following is a trivial example of a selector generator:

```
my_record = datatype{unordered FL} is get_{FL}

    rep = record{FL}

    foreach {F: T in FL} get_ = proc(r: cvt) returns(T)
        return(r.F)
    end

end
```

Inside a selector generator the formal field list declared in the datatype header (parameter FL above) is handled by special language constructs. These constructs declare a formal field name and a formal field type that can be used inside the construct. At datatype instantiation the generator is given an actual field list and code for each name-type pair in the field list is generated from these special language constructs by replacing occurences of the formal field names and types with the actual ones. In the example a `get_`*field_name* procedure to fetch the corresponding component (*field_name*) of the argument is generated for each name-type pair supplied for the generator, and we obtain:

get_name = **proc**(r: **cvt**) **returns**(string)
...
get_age = **proc**(r: **cvt**) **returns**(int)
...

Thus it is possible for the user to write his own record generator, if he is not contented with the built-in one. Selector generators also made it possible to write the whole built-in library of XE in the language itself by using a few simple primitives implemented in the code generator. Thus portability is enhanced and the built-in library benefits from any advance in code optimization.

When data types are designed the number of operations included in them should usually be kept small, so that only the operations really essential to the type are defined. The operations are named according to the concepts of the abstraction that the type implements. Often type generators or other parameterized modules make restrictions on their type parameters that exclude a number of types that would be semantically appropriate. Therefore XE has a mechanism to insert (and rename) operations into a type at instantiation time. For example, if a priority queue generator requires an operation `better` and the `int` type does not have it, the following priority queue instance can be still made:

int_queue = priority_queue[int **use** better = int$gt]

A similar construct has been used in Argus [Lisk87], which is also related to CLU.

4. Type System and Built-in Types

The type system of XE differs slightly from the type system of CLU. In both languages a routine type specification can include a list of exceptional conditions that can be signalled by a routine of that type. In CLU a routine can be assigned to a variable if the variable has the same type as the routine or if the variable is of the type **any**. In other words the set of exceptions (signals) must be equivalent. In XE there is a special type inclusion of routine types – also those routines that have a subset of the declared signals are assignment compatible. It is even possible to define routine types that include all signals. This kind of definitions are often needed in parameter restrictions (**where** clauses) and have been found to be very useful.

In the following example the sum of elements of a container is computed. The addition operation `add` of the element type is allowed to signal anything. The computation is aborted if a signal is raised by `add`. The name of the signal is then signalled to the invoker.

```
sum = proc[t, e: type](tuple: t) returns(e)
          signals(empty, not_possible(string))
      where t has elements: itertype(t) yields(e)
      where e has add: proctype(e, e) returns(e) signals(*)
   empty: bool := true
   result: e
   for element: e in elements(:tuple) do
      if empty then
         result := element
         empty := false
      else
         result := result + element
            except otherwise(why: string):
               signal not_possible(why)
            end
      end
   end
   if empty then
      signal empty
   end
   return(result)
end
```

The built-in libraries of CLU and XE are different. CLU has a special type **any** that is the union type of all types. According to our experience this type is of very little use, and therefore it was not included in XE. The absence of the **any** type also enables all type checking to be performed at compile time.

The other most important differences in the built-in library are arrays of static size (in CLU they are dynamic), modular arithmetics, the presence of an enumeration type generator `tag`, and the absence of the real numbers. Dynamic arrays and arithmetic operations that signal overflow conditions were considered too expensive for embedded systems.

The operations of the built-in types of XE are designed in an uniform way. All operations with a similar function have the same name and a similar interface. For example, all built-in types of XE have `parse` and `unparse` operations. They construct objects from a textual form and print objects into the same textual form. The `parse` operations perform a syntax analysis of the argument string. If an error occurs they signal an exception that contains an error message and position of the error in the argument. For example, if the user has written a data type `expression` that has the `parse` operation, he can use it in the following way to read input, where each input line can be an expression list:

```
read_expr_list = proc(i, o: stream, prompt: string) returns(array[expression])
   while true do
      puts(:o, prompt)
      return(array[expression]$parse(getl(:i)))
         except when syntax_error(msg: string, pos: int):
            for in [1 .. pos + size(:prompt)] do
               putc(:o, ' ')
            end
            putl(:o, "^")
            putl(:o, msg)
         end
   end
end
```

The example above reads a list of expressions (e.g., `[1 + 2, x - c, d]`) from the input, and returns an array of corresponding expression objects. The parse operation of the built-in type generator array invokes the parse operation of its parameter type for each element. If the input is erroneous, an error message is given and the position of the error is indicated by a ^ character.

Often in the design of data types some operations of the representation type can be used as they are. For example, the `parse`, `unparse`, `copy`, `equal` and `similar` operations of a representation type often have the semantics needed by the abstraction without any modifications. In XE operations can be inherited from the representation type. This kind of inheritance is quite useful, even though it is less powerfull than the inheritance mechanisms used in object-oriented languages. Using inheritance the example datatype generator `my_record` can be written:

```
my_record = datatype{unordered FL} inherits get_{FL}

   rep = record{FL}

end
```

5. Other Features

Procedures written in other languages can be invoked from XE. Typically the major problem with such foreign procedures is the argument and return value passing because of different representations of data. In XE the argument objects are converted by `dump` operations of their types before passing them to the foreign procedure invoked. The result objects produced by the foreign procedure are converted into the XE representation by invoking `undump` operations of their XE types. Thus it is possible to use all types that have the `dump` and `undump` operations (including user-defined types) in the communication with the outside world. These operations designed for efficient communication of binary data are provided for all built-in XE types and type generators.

The iteration mechanism of CLU is quite useful, but it is not able to cope with situations where there is a need for several simultaneous iterations that are not nested within each other. The **iterate** statement of XE executes a set of iterators in parallel as suggested in [Ecka87]. Thus it is possible, for example, to write a merge sort using iterators:

```
merge = proc[t: type](a1, a2: array[t]) returns(array[t])
        where t has gt: proctype(t, t) returns(bool)
   result: array[t] := array[t]$alloc(size(:a1) + size(:a2))
   iterate i1: e1: t in elements(:a1)
           i2: e2: t in elements(:a2) do
      for i: int in [0 .. size(:result) - 1] do
         if ~terminated i2 cand (terminated i1 cor e1 > e2) then
            result[i] := e2
            next i2
         else
            result[i] := e1
            next i1
         end
      end
   end
   return(result)
end
```

Besides the operation overloading of CLU XE has overloading for literals. Each literal form is connected with a data type operation. If a data type has the corresponding operation, the literal can be used to denote its objects. For example, the user can write his own integer type `my_int` and still use constructs like

```
x: my_int := 5
```

if he provides the type with the `create` operation for integer literals.

In addition to the semantic differences discussed above there exist quite a lot of minor differences between XE and CLU. Some of the differences are purely syntactical. These differences were sometimes dictated by implementational facts (CLU uses a predictive parser, XE an LALR parser).

The design of XE was based on CLU, and XE resembles it in many ways, but there are major differences between the languages. The XE compiler cannot compile

CLU programs – XE is not an extension of CLU. Many cases that can be handled abstractly in XE are quite hard to write in CLU. We feel that our design has yielded a new useful programming language.

We have implemented an integrated programming environment for XE. The first XE application is running: XEDA, a 5000 line embedded diagnostic advisor of a digital telephonic exchange.

Acknowlegdement

The Exbed project was funded by the Technology Development Centre, Nokia Corporation, and KONE Corporation.

References

[Arkk88] Arkko, J., Hirvisalo, V., Kuusela, J., Nuutila, E., and Tamminen, M., *Some Experiences with Rules in Procedural Programming Languages*, Proc. International Computer Science Conference '88, Hong Kong, Dec. 19-21 1988, pp. 712-718.

[Arkk89] Arkko J., Hirvisalo V., Kuusela J., Nuutila E., Tamminen M.: *XE Reference Manual*, Helsinki University of Technology, Report HTKK-TKO-A26, 1989, 109p.

[Ecka87] Eckart J.D.: *Iteration and Abstract Data Types*, ACM Sigplan Notices, Vol. 22, No. 4, April 1987, pp. 103-110.

[Lisk81] Liskov B., et al.: *CLU Reference Manual*, Springer-Verlag, 1981, 190p.

[Lisk87] Liskov B. et al.: *Argus Reference Manual*, MIT, Programming Methodology Group Memo 54, 1987, 187p.

[Nuut87] Nuutila, E., Kuusela, J., Tamminen, M., Veilahti, J., Arkko, J. and Bouteldja, N., *XC – A Language for Embedded Rule Based Systems*. SIGPLAN Notices, Vol. 22, No. 9, pp. 23-32, 1987.

[Stro86] Stroustrup B.: *The C++ Programming Language*, Addison Wesley, 1986, 327p.