Concrete Syntax for Data Objects in Functional Languages

Annika Aasa Kent Petersson Dan Synek

Programming Methodology Group
Dept. of Computer Science,
Univ. of Göteborg and Chalmers,
S-412 96 Göteborg,
Sweden

1 Introduction

Many functional languages have a construction to define inductive data types [Hoa75] (also called general structured types [Pey87], structures [Lan64], datatypes [Mil84] and free algebras [GTWW77]). An inductive definition of a data type can also be seen as a grammar for a language and the elements of the data type as the phrases of the language. So defining an inductive data type can be seen as introducing an embedded language of values into the programming language. This correspondence is however not fully exploited in existing functional languages. The elements can presently only be written in a very restricted form. They are just the parse trees of the elements written in prefix form. A generalization, that we will consider in this paper, is to allow the elements to be written in a more general form. Instead of directly writing the parse trees of the embedded language, we would like to use a more concrete syntactical form and let an automatically generated parser translate the concrete syntactical form to the corresponding parse tree. We think that this is especially useful when we manipulate languages in programs, for example, when implementing compilers, interpreters, program transformation systems, and programming logics. It is also convenient if we want to use the concrete syntax for other kinds of data in a program.

By allowing distfix operators in a programming language [Pey86], it is possible to achieve some of the goals we have presented above. The problem is that the symbols comprising the distfix operator must not interfere with the constructions of the programming language itself. If we want to represent programs of a language in the language itself, this problem becomes acute. For example, to represent arithmetic expressions inside a functional language, it is difficult, but not impossible, to let 'x+23' in one situation be an expression which evaluates to an integer and in another a value that represents an arithmetic expression. We can solve this problem in at least two different ways. We can either say that the distfix operator must be built up from identifiers of the programming language or we can make a clear distinction between the programming language, the *metalanguage*, and the represented language, the *object language*. Of course we can relax the situation in the first case a little by allowing overloaded identifiers and operators in the metalanguage, but it is hard to imagine how pure syntactical constructions of the metalanguage, for example reserved words, could be overloaded.

2 Concrete Data Types

We start by introducing a syntactical construction into our favorite functional language (ours is ML), to define a concrete data type of binary numbers[1] as:

```
conctype BinNumber =  [|0|]
                   |  [|1|]
                   |  [|<BinNumber>0|]
                   |  [|<BinNumber>1|]
```

Compare the type definition with the context free grammar for binary numbers

```
<BinNumber> ::=  0
              |  1
              |  <BinNumber>0
              |  <BinNumber>1
```

Since we must not confuse the symbols of the defined language (the Object Language – OL) with the symbols of the programming language (the Meta Language – ML), we enclose the elements in *quotation brackets*, `[|...|]`. Notice that the nonterminals in the grammar correspond to types in the type definition. The intention is to introduce a data type for binary numbers and let the elements be written in the familiar way. So in a program we would like to write the elements as `[|1011|]` and `[|1010010101|]`. We use the name *quotation expression* for this new form of expression.

We also want to be able to define computations over the elements, so we need a construction that separates the different forms a binary number can take and selects the components of a particular form. The modern way to do this in a functional language is to use pattern matching. We therefore introduce a pattern matching form for the elements of a type defined by our new constructor. A pattern is a sentential form of the language defined by the concrete datatype with ordinary ML-patterns of type A instead of nonterminals A. Since our patterns may contain ML patterns, we use an *antiquotation symbol*, '`^`', to write ML variables and patterns in the object language phrases. Variables can be written just following the antiquotation symbol but more complicated patterns must be enclosed in parentheses. Blanks after a variable are ignored. Examples of patterns for the concrete datatype of binary numbers are `[|1011|]`, `[|^x|]`, `[|^x 1011|]` and `[|^([|10|])1|]`. We use the name *quotation pattern* for this new form of pattern. By using this construction it is possible to write a function that takes a binary number as argument and gives its successor as result:

```
fun succ [|0|]      = [|1|]
|   succ [|1|]      = [|10|]
|   succ [|^b 0|]   = [|^b 1|]
|   succ [|^b 1|]   = [|^(succ b)0|]
```

Notice that we have used the antiquotation symbol also in the quotation expressions in the right hand sides of the function definition. In the example, there is first a quotation expression `[|^b 1|]` which is intended to construct a phrase from the value bound to the ML-variable `b` and the symbol `1`. The variable must, of course, be bound to a value of type `BinNumber` since such a value is expected in this position. Secondly, there is the more complicated expression `[|^(succ b)0|]` where the ML-expression `succ b` is evaluated to a value of type `BinNumber` and this number is then composed with the symbol `0` to produce a binary number. All ML-expressions inside a quotation must evaluate to complete phrases of the language and not to strings of characters.

Patterns in a function definition need not directly correspond to the cases in the definition of the concrete datatype, as can be seen by the following example:

```
fun div4 [|^x 00|] = true
|   div4 y         = false
```

The pattern matching using the concrete syntax is important in our approach to representing object languages. Without it, one has to introduce somewhat arbitrary names for the operations that decides what form an element has and for the operations that selects the components of a compound element. Compare our quotation brackets and antiquotation symbol with the corresponding constructions in the Edinburgh LCF system [GMW79] and also with Quine's quasi-quotation [Qui81].

As a second example consider implementing the denotational description of a very simple imperative language. We first define the language

```
conctype Pgm
   =  [|program <Cmds> end|]
and Cmds
   =  [|<Cmd>|]
   |  [|<Cmd>; <Cmds>|]
and Cmd
   =  [|if <Bexp> then <Cmds> else <Cmds> fi|]
   |  [|while <Bexp> do <Cmds> end|]
   |  [|<Var>:=<Exp>|]
and Exp
   =  [|<Var>|]
   |  [|<Integer>|]
   |  [|<Exp>+<Exp>|]
   |  [|<Exp>*<Exp>|]
   |  [|(<Exp>)|]
```

```
and Bexp
   =  [|<Exp>=<Exp>|]
   |  [|<Bexp>|<Bexp>|]
   |  [|<Bexp>&<Bexp>|]
   |  [|(<Bexp>)|]
```

where we assume we already have defined two concrete datatypes `Var` and `Integer` and a function `toint` that maps a concrete integer to the corresponding ML value. Using the concrete datatype defined above, we can define an interpreter in a natural way. Notice that the definition is very close to how Gordon describes the denotational semantics of a language in [Gor79]. We assume we already have implemented an abstract data type for states, with operations `sinit`, `update` and `valof`.

```
fun P  [|program ^(cs) end|] = Cs cs sinit
and Cs [|^c|] s      = C c s
|   Cs [|^c; ^cs|] s = Cs cs (C c s)
and C  [|if ^(b) then ^(s1) else ^(s2) fi|] s =
       if B b s then Cs s1 s else Cs s2 s
|   C  [|while ^(b) do ^(cs) end|] s =
       let fun f s = if B b s then f (Cs cs s)
                                   else s
       in f s
       end
|   C  [|^x:=^e|] s =
          update(x, E e s, s)
and E  [|^x|] s        = valof(x,s)
|   E  [|^n|] s        = toint n
|   E  [|^e1+^e2|] s   = E e1 s + E e2 s
|   E  [|^e1*^e2|] s   = E e1 s * E e2 s
|   E  [|(^e)|] s      = E e s
and B  [|^e1=^e2|] s   = E e1 s = E e2 s
|   B  [|^b1|^b2|] s   = B b1 s orelse  B b2 s
|   B  [|^b1&^b2|] s   = B b1 s andalso B b2 s
|   B  [|(^b)|] s      = B b s
```

This example raises a problem of how to decide what different patterns means. The first and second cases in the definition of `E` just consist of ML-variables and the problem is how to decide that the first is the variable case and the second the integer case. We use information from the type inference mechanism to choose between the two possibilities. The pattern is not parsed until the typechecker already has typechecked the right hand sides of the definition and then we know that `x` in the first case must be of type `Var` and `n` in the second must be of type `Integer`. From this information it is possible to distinguish which case a pattern is supposed to denote. It is of course possible to try to define a function where one can not decide what cases two patterns are supposed to denote. Take for example the definition of a function that counts the number of variables in an expression:

```
fun Vars [|^x|]        = 1
|   Vars [|^n|]        = 0
|   Vars [|^e1+^e2|]   = Vars e1 + Vars e2
|   Vars [|^e1*^e2|]   = Vars e1 + Vars e2
|   Vars [|(^e)|]      = Vars e
```

In this example it is impossible to choose between the two cases if we do not use the variable name to indicate its type, as we do in denotational descriptions and Fortran(!!). Our solution is to allow the user to explicitly type the variables. So the first two cases in the definition must be written as:

```
fun Vars [|^(x:Var)|]      =  1
|   Vars [|^(n:Integer)|]  =  0
...
```

We consider it to be an error if we do not have enough information to parse a quotation pattern unambiguously.

We have another problem if we want to define a function

```
fun isadd [|^x+^y|] = true
|   isadd z         = false
```

because the type checker and quotation parser can not give a unique type to the variables `x` and `y`. They can either be of type `Var`, `Integer` or `Exp`, so the pattern is ambiguous and therefore erroneous. To make it unambiguous the user must provide type information. Notice that the type information distinguishes the more restrictive pattern `[|^(x:Var)+^(y:Var)|]` from `[|^(x:Exp)+^(y:Exp)|]`. Problems with ambiguities in patterns are discussed in a paper [DKLM84].

The concrete data types fit nicely into the ordinary typesystem in ML and we can for example define polymorphic concrete data types such as trees with information in the nodes.

```
conctype 'A Tree = [|o|]
                 | [|{<'A Tree>-<'A>-<'A Tree>}|]
```

with elements like

```
[|{o-^("HEJA")-{o-^("BARACKEN")-o}}|]
: String Tree
```

and

```
[|{o-1010-o}|] : BinNumber Tree
```

and a function that swaps the left and right part of a tree

```
fun swaptree [|o|]             = [|o|]
|   swaptree [|{^x-^y-^z}|] = [|{^z-^y-^x}|]
```

As can be seen from the String Tree example above, it is possible to use ordinary ML types when defining concrete types.

3 Lexical Analysis

It is not obvious what should be treated as a lexical token in the embedded languages. In order to be flexible and allow as many and as different concrete data types as possible, we have decided to view every character as a lexical token. The only exceptions to this are that a sequence of blanks is treated as one blank and that the escape character '\' gives the following character its literal meaning. The result of this is that blanks are not handled nicely. If we want to have blanks in a quotation expression then there must be a blank character in the corresponding position in the grammar, and if a blank is present in the grammar there must be at least one blank in the quotation.

Having a more sophisticated lexical analyzer give us another problem. We can not use parts of a lexical token in the grammar. For example if we use ML:s lexical analyzer, as they do in the LeML system [The85], we can not define the binary numbers as we do in section 2 since a sequence of zeros and ones is treated as an integer in ML.

The best solution would probably be to give the user the possibility to define her own lexical analyzer.

4 Parsing and Type Derivation

In this section we describe how the new constructions are translated during the compilation to ordinary data types and constructors.

After the compilation nothing of the new constructions remains and they have therefore no effect on the execution speed of the new syntactical constructions. A program with concrete datatypes, quotations and anti-quotations runs at the same speed as one without them.

Let us give an overview of the translation process. A concrete datatype is translated into the following objects.

- A datatype definition, which could be seen as the type of parse trees for the language defined by the grammar in the concrete datatype.
- A parser that recognizes the language described by the grammar and translates a phrase of the language into the parse tree.
- A (pretty?) printer that prints elements of the concrete datatype using the concrete syntax.

The type definition of binary numbers,

```
conctype BinNumber =  [|0|]
                   |  [|1|]
                   |  [|<BinNumber>0|]
                   |  [|<BinNumber>1|]
```

will be translated into the following datatype definition.

```
datatype BinNumber = BinNumber1
                   | BinNumber2
                   | BinNumber3 of BinNumber
                   | BinNumber4 of BinNumber
```

and a parser that, for example, translates [|101|] to BinNumber4(BinNumber3(BinNumber2))

The parser is used in the compiler to translate quotation patterns and quotation expressions to ordinary patterns and expressions. For example, the function

```
fun succ [|0|]      = [|1|]
|   succ [|1|]      = [|10|]
|   succ [|^b 0|]   = [|^b 1|]
|   succ [|^b 1|]   = [|^(succ b)0|]
```

is translated to

```
fun succ (BinNumber1)   = BinNumber2
|   succ (BinNumber2)   = BinNumber3 BinNumber2
|   succ (BinNumber3 b) = BinNumber4 b
|   succ (BinNumber4 b) = BinNumber3 (succ b)
```

Usually the syntax of an ML program is checked in two distinct steps. First it is parsed with a context free parser and then typechecked with a type checker utilizing Milner's algorithm [Mil78]. This simple sequence is no longer possible when concrete data types are added to ML since the parsing of a concrete element depends of its type. That is, if a concrete expression is in a context where an element of type A is expected it should be parsed with A as the start symbol. Correspondingly, the type of an ML expression in a quotation is dependent on the parsing of the quotation. To achieve this, we have to integrate the parsing of the concrete elements with Milner's type derivation algorithm. We have built the parser around a generalized version of Earley's algorithm [Ear70]. It is generalized for two reasons:

- Concrete data types seen as grammars are more powerful than context free grammars for which Earley's algorithm is constructed. This stems from the fact that we can have polymorphic concrete data types. A simple example of a language that can be defined by a polymorphic data type but not with a context free grammar are the trees introduced in section 2.
- A parser defined with Earley's algorithm usually takes a sequence of lexical tokens as input and gives a parse tree as output. However, the parser that we want should take a sequence of lexical tokens *and unquoted ML objects* as input and give an ML expression as output.

The algorithm we have developed differs from Earley's in that we can have type variables in the nonterminals and these can be instantiated during parsing. For example, given the conctype

```
conctype 'A List = [|$|]
                 | [|<'A><'A List>|]
and
conctype D       = [|0|] | [|1|]
```

and the input `[|0$|]`, `'A` will be instantiated to `D`. To support this, each item in Earley's algorithm contains, apart from the dotted production and the item set pointer, a type substitution. The substitutions are handled in the following way by the parsing operations:

- In the predict operation in Earley's algorithm an item is added only once even though it might be predicted from more than one item. In our version each new item is created with an initial substitution. The more information we let this substitution inherit from the predicting item the less is the chance that we can share it. To assure maximal sharing we let each item start with the empty substitution. When the dot is to the left of an uninstantiated type variable all conctypes known in the context are predicted.
- In the completion of an item I, we return to the item set pointed to and add updated versions of all items I' which have a type T' to the right of the dot that can be unified with the type T we have just completed. Since there can be more than one item with this property we have to make sure that we use fresh variables for the type variables in T before unification. We update I' by moving the dot one step to the right and compose the substitution with the substitution in I.

Let us illustrate this with an example: Corresponding to the input `[|$$|]` and the conctype

```
conctype 'A List = [|$|]
                 | [|<'A><'A List>|]
```

we get the item sets:

$$\begin{array}{lcll}
 & & I_0 & \\
<\mathrm{S}> & ::= & \cdot < 'A \text{ List} >,\ s_0,\ 0 & (1)\\
< 'A \text{ List} > & ::= & \cdot \$,\ [],\ 0 & \\
< 'A \text{ List} > & ::= & \cdot < 'A > < 'A \text{ List} >,\ [],\ 0 &
\end{array}$$

$$\begin{array}{lcll}
 & & I_1 & \\
< 'A \text{ List} > & ::= & \$ \cdot,\ [],\ 0 & (2)\\
<\mathrm{S}> & ::= & < 'A \text{ List} > \cdot,\ s_0['A \mapsto 'X_1],\ 0 & (3)\\
< 'A \text{ List} > & ::= & < 'A > \cdot < 'A \text{ List} >, & \\
 & & ['A \mapsto 'X_2 \text{ List}],\ 0 & \\
< 'A \text{ List} > & ::= & \cdot \$,\ [],\ 1 & \\
< 'A \text{ List} > & ::= & \cdot < 'A > < 'A \text{ List} >,\ [],\ 1 &
\end{array}$$

$$I_2$$

$$\begin{array}{lll}
\langle\, 'A \text{ List}\,\rangle & ::= & \$\cdot,\ [],\ 1 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \langle\, 'A\,\rangle \langle\, 'A \text{ List}\,\rangle \cdot, \\
 & & ['A \mapsto 'X_2 \text{ List}, \\
 & & \quad 'X_3 \mapsto 'X_2 \text{ List}],\ 0 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \langle\, 'A\,\rangle \cdot \langle\, 'A \text{ List}\,\rangle, \\
 & & ['A \mapsto 'X_4 \text{ List}],\ 1 \\
\langle S \rangle & ::= & \langle\, 'A \text{ List}\,\rangle \cdot s_0['A \mapsto 'X_5, \\
 & & \quad 'X_5 \mapsto 'X_2 \text{ List}, \\
 & & \quad 'X_3 \mapsto 'X_2 \text{ List}],\ 0 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \langle\, 'A\,\rangle \cdot \langle\, 'A \text{ List}\,\rangle, \\
 & & ['A \mapsto 'X_6 \text{ List}, \\
 & & \quad 'X_6 \mapsto 'X_2 \text{ List}, \\
 & & \quad 'X_3 \mapsto 'X_2 \text{ List}],\ 0 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \cdot\$,\ [],\ 2 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \cdot \langle\, 'A\,\rangle \langle\, 'A \text{ List}\,\rangle,\ [],\ 2
\end{array}$$

The composition of substitutions s and s' is written ss'. s_0 is the substitution obtained from Milner's type derivation algorithm applied to the ML parts of the program.

Notice that before we unify the left hand side of the production in (2) with the type to the right of the dot in (1) to get (3) we substitute a new variable $'X_1$ for $'A$ in (2).

We now explain what to do with unquoted expressions in the input. An unquoted expression of type T can be viewed as an already parsed part of the input that can be accepted whenever we have an item in the item set with the dot to the left of a type that can be unified with T. We illustrate this with the same grammar as above and the input `[|^(x:Int)$|]`

$$I_0$$

$$\begin{array}{lll}
\langle S \rangle & ::= & \cdot \langle\, 'A \text{ List}\,\rangle,\ s_0,\ 0 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \cdot\$,\ [],\ 0 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \cdot \langle\, 'A\,\rangle \langle\, 'A \text{ List}\,\rangle,\ [],\ 0
\end{array}$$

$$I_1$$

$$\begin{array}{lll}
\langle\, 'A \text{ List}\,\rangle & ::= & \langle\, 'A\,\rangle \cdot \langle\, 'A \text{ List}\,\rangle,\ ['A \mapsto \text{Int}],\ 0 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \cdot\$,\ [],\ 1 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \cdot \langle\, 'A\,\rangle \langle\, 'A \text{ List}\,\rangle,\ [],\ 1
\end{array}$$

$$I_2$$

$$\begin{array}{lll}
\langle\, 'A \text{ List}\,\rangle & ::= & \$\cdot,\ [],\ 1 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \langle\, 'A\,\rangle \langle\, 'A \text{ List}\,\rangle \cdot,\ ['A \mapsto \text{Int}, \\
 & & \quad 'X_1 \mapsto \text{Int}],\ 0 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \langle\, 'A\,\rangle \cdot \langle\, 'A \text{ List}\,\rangle, \\
 & & ['A \mapsto 'X_2 \text{ List}],\ 1 \\
\langle S \rangle & ::= & \langle\, 'A \text{ List}\,\rangle \cdot,\ s_0['A \mapsto 'X_3, \\
 & & \quad 'X_3 \mapsto \text{Int}, \\
 & & \quad 'X_1 \mapsto \text{Int}],\ 0 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \langle\, 'A\,\rangle \cdot \langle\, 'A \text{ List}\,\rangle,\ ['A \mapsto \text{Int List}, \\
 & & \quad 'X_4 \mapsto \text{Int},\ 'X_1 \mapsto \text{Int}],\ 0 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \cdot\$,\ [],\ 2 \\
\langle\, 'A \text{ List}\,\rangle & ::= & \cdot \langle\, 'A\,\rangle \langle\, 'A \text{ List}\,\rangle,\ [],\ 2
\end{array}$$

5 User Defined Representation

In the previous section we have described how the elements of the concrete datatypes are represented. Sometimes this representation is a bit inconvenient because we often want to represent sequences as ML lists and sometimes we would like to "forget" some productions in the grammar. These differences in representation should be indicated by constructions in the definition of conctypes.

Productions containing terminals whose only purpose are to indicate the structure of a sentence are of course unnecessary when the sentence is represented as a tree. Take for example parentheses in arithmetic expressions. In the denotational description in section 2 we have the clauses `[|(<Exp>)|]` and `[|(<Bexp>)|]` because we want to have parentheses in elements of the concrete datatypes `Exp` and `Bexp`. These productions remain in the representation and we must therefore have the cases:

```
E [|(^e)|] s = E e s
B [|(^b)|] s = B b s
```

in the interpreter. To eliminate these let us introduce the notation

```
A = [||...<A>...||]
```

to indicate that the production should not remain in the representation. The dots stands for arbitrary terminal symbols. Our denotational example would then be written:

```
  Exp = ...
      | [||(<Exp>)||]
 Bexp = ...
      | [||(<Bexp>)||]
```

The parentheses may only occur in quotation expressions. They are eliminated during parsing and must not appear in quotation patterns. The only situation when it is possible to forget productions in this way is of course when the clause just contains one nonterminal and this is the same as the conctype just being defined.

In grammars, we often use a special notation for sequences.

$$Cmds ::= Cmd \ \{; Cmd\}^*$$

In the conctypes described so far we have used recursion to define sequences. For example in the denotational description in section 2 where a sequence of commands is defined by:

```
conctype Cmds = [|<Cmd>|]
              | [|<Cmd>; <Cmds>|]
```

A sequence of commands `c1;c2;c3` is represented by the term:

```
Cmds2(c1,Cmds2(c2,Cmds1 c3))
```

Having this representation we must use explicit recursion when defining computations over the elements. If we instead represent the sequences as ML lists it is possible to use predefined list handling functions. When representing sequences as ML lists it is necessary to exclude the terminals which separate the elements. We therefore use a somewhat different notation for sequences than in ordinary grammar descriptions.

```
{<Cmd>; ...}+
```

The nonterminal inside the curly brackets defines the elements in the sequence. The terminals between the nonterminal and the three dots are the separators between the elements in the sequence. We use + to denote repetition one or more times. It is also possible to use * to denote zero or more times. Using this notation the command sequence in the denotational description could be defined

```
conctype Pgm = [|program {<Cmd>; ...}+ end|]
```

The conctype `Cmds` is then no longer necessary. Neither is the function `Cs` in the interpreter. Instead we define the function `P` thus

```
fun P [|program ^(cs) end|] = foldleft C sinit cs
```

where `foldleft` is a predefined list handling function.

The conctype `Pgm` is translated to the datatype:

```
datatype Pgm = Pgm1 of Cmd list
```

Using list notation, the definition of a language by conctypes becomes more comprehensible. A procedure head in a simple imperative language could with list notation be defined as:

```
conctype procedurehead
   = [|PROC <Id>;|]
   | [|PROC <Id>({<Parlist>;...}+);|]

and Parlist
   = [|{<Id>,...}+:<TypeId>|]
```

where `Id` and `Typeid` are two already defined conctypes. An element of this type is `[|PROC P(a,b:int;ch:char);|]` and a function that counts the number of parameters is:

```
fun countpars [|PROC ^id;|]        = 0
|   countpars [|PROC ^id (^ps);|] =
    sum (map (fn [|^ids:^t|] => (length ids)) ps)
```

where `sum`, `map` and `length` are predefined list handling functions. Without list notation the conctype definition must be defined as:

```
conctype Prochead
   = [|PROC <Id>;|]
   | [|PROC <Id>(<Parlist>);|]

and Parlist
   = [|<Idlist>:<Typeid>|]
   | [|<Idlist>:<Typeid>;<Parlist>|]

and Idlist
   = [|<Id>|]
   | [|<Id>,<Idlist>|]
```

and the function that counts the number of parameters:

```
fun countpars [|PROC ^id;|]       = 0
|   countpars [|PROC ^id(^ps);|] = countpar ps

fun countpar [|^ids:^t|]      = countids ids
|   countpar [|^ids:^t;^ps|] =
        countids ids + countpar ps

fun countids [|^id|]        = 1
|   countids [|^id,^ids|] = 1 + countids ids
```

As previously mentioned, it is considered to be an error if a quotation expression can not be parsed uniquely. It is therefore desirable to have unambiguous conctypes. Consider arithmetic expressions. An unambiguous conctype would contain a lot of nonterminals and productions which are irrelevant in the representation. Many function definitions thus become quite complicated. Using precedences and associativity rules which resolve ambiguities, we can define the same language with a less complicated conctype, and have more natural patterns. A desirable extension is therefore to give the user the possibility of giving precedence and associativity rules, to productions in the conctypes.

A completely different representation could be obtained by defining a datatype and a function that maps the concrete object to the new representation. If we for example want to represent binary numbers as integers we could define a function `bintoint` which maps the concrete data type `BinNumber` to the corresponding integer.

```
fun bintoint [|0|]     = 0
|   bintoint [|1|]     = 1
|   bintoint [|^x 0|] = 2 * bintoint x
|   bintoint [|^x 1|] = 2 * bintoint x + 1
```

Compare this with the following definition in YACC.

```
binnumb: ZERO           {$$ = 0;}
|        ONE            {$$ = 1;}
|        binnumb ZERO   {$$ = $1*2;}
|        binnumb ONE    {$$ = $1*2+1;}
```

6 Implementation

The constructions we described in section 2 have been implemented in the functional language LML [AJ87,Aug84] and all examples in that section have been tested in the implementation. The constructions described in section 5 are not implemented yet.

LML has no input and output for user defined datatypes so we have not bothered to generate pretty printers for conctypes in our implementation.

7 Related Work

One system with an explicit notion of metalanguage and object language is the Edinburgh LCF system [GMW79]. In contrast to our proposal, LCF contains only one fixed object language. Furthermore, the object language is represented by an abstract type that defines the abstract syntax of the language, so the concrete syntax in quotations is seen just as a convenient way for the user to enter elements of this type. To define computations that uses the object language one has to use the constructors and selectors of the abstract syntax and the user must therefore remember both the concrete and the abstract syntax of the object language.

In the LeML system from INRIA [The85,Hue86], the user can easily define his own object language by using an interface with an ML version of YACC. But the concrete syntax must still be seen as a convenient form to write abstract syntax trees since all computations must be expressed in terms of the constructors and selectors of the abstract syntax. Nothing like our quotation patterns is available. Wand [Wan84] has implemented a similar system for Scheme also using YACC to generate the parser that translates from concrete to abstract syntax.

A more limited way to define an object language is to use infix operators as constructors, as we can do in ML [Mil84]. A type declaration for arithmetic expressions involving integers and the operators + and * can in ML be defined as

```
infix ++ **
datatype Expr = NUM of int
              | op ++ of Expr * Expr
              | op ** of Expr * Expr
```

A function which evaluates such an expression is:

```
fun E (NUM n)     = n
|   E (e1 ++ e2) = E e1 + E e2
|   E (e1 ** e2) = E e1 * E e2
```

This way of making the elements of a datatype more concrete has a number of disadvantages. Since ML does not allow overloading we can not use the symbols + and * as constructors. We have to choose other symbols, for example ++ and ** as in the example above. We must also have a constructor for each part of the new language we want to define, even for those parts which do not have a constructor in the concrete syntax, like the integer case in the example above. The expression `2*3+4` must therefore be written `(NUM 2)**(NUM 3)++(NUM 4)`. If we want to write expressions in a more familiar way we must write a parser which translates strings to elements in the datatype.

8 Future Work

In the future we will implement the constructions described in the section on user defined representation. We will also define and implement constructions for expressing priorities and associativity. The problems with the lexical analyzer also have to be further investigated. Another interesting question is if the notion of subtype [FM88] could resolve some of the problems with ambiguities in patterns we have described.

Acknowledgements

We would like to thank the members of the Programming Methodology Group in Göteborg for their help and encouragement. In particular we would like to thank Lennart Augustsson for his support and all help he provided during the implementation and Sören Holmström for his useful comments.

References

[AJ87] L. Augustsson and T. Johnsson. *Lazy ML User's Manual.* Programming Methodology Group, Department of Computer Sciences, Chalmers, S–412 96 Göteborg, Sweden, 1987. To be distributed with the LML compiler.

[Aug84] L. Augustsson. A compiler for lazy ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, 1984.

[DKLM84] V. Donzeau-Gouge, G. Kahn, B. Lang, and B Mélèse. Document structure and modularity in Mentor. In *Proceedings of the ACM SIGSOFT/SIGPLAN - Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, 1984. Software Engineering Notes Vol. 9, No 3.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In H. Ganzinger, editor, *Proceedings ESOP '88. LNCS vol. 300*, pages 94–114, Springer-Verlag, Nancy, France, 1988.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[Gor79] M. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[GTWW77] J. A. Gougen, J. W Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *JACM*, 24(1):68–95, January 1977.

[Hoa75] C. A. R. Hoare. Recursive data structures. *International Journal of Computer and Information Sciences*, 4(2):105 – 132, 1975.

[Hue86] G. Huet. Formal structures for computation and deduction. May 1986. Lecture Notes for International Summer School on Logic Programming and Calculi of Discrete Design, Marktoberdorf, Germany.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[Mil84] R. Milner. Standard ML proposal. *Polymorphism: The ML/LCF/Hope Newsletter*, 1(3), January 1984.

[Pey86] S.L. Peyton Jones. Parsing distfix operators. *Communications of the ACM*, 29(2):118–122, February 1986.

[Pey87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Qui81] Willard Van Orman Quine. *Mathematical Logic.* Harward University Press, 1981.

[The85] The ML handbook, version 6.1. Project Formel, Inria, May 1985.

[Wan84] Mitchell Wand. A semantic prototyping system. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 213–221, June 1984.