RAL-90-067

Science and Engineering Research Council
Rutherford Appleton Laboratory
Chilton DIDCOT Oxon OX11 0QX

RAL-90-067

The Incremental Development of an Algorithm for Matching with Higher Order Variables

J C Bicarregui and B M Matthews

August 1990

The Incremental Development of an Algorithm for Matching with Higher Order Variables

J.C.Bicarregui and B.M.Matthews
Rutherford Appleton Laboratory
Chilton, DIDCOT,
Oxon OX11 0QX.

Abstract

We describe the problem of pattern matching with higher order variables as it arises in the context of theorem proving in a natural deduction style. The need for a matching algorithm arises when constructing a justification of lines in proofs. The algorithm produces the possible instantiations of meta-variables in the rule that lead to a match with the proof line. The higher order nature of the matching refers to the possibility of having parameterised meta-variables in the inference rules of the system. The higher order meta-variables considered here are not as general as those occurring in the lambda calculus: here syntactic identity is required in the instantiated solutions rather than incorporating β-equivalence as primitive.

In order to address the development of the algorithm, successively specialised cases of the problem are considered. At each stage formal specifications of the key functions are given in VDM and then developed in the functional language Standard ML. The algorithm described by the ML code has been used as a specification for a part of a larger theorem proving system implemented imperatively.

1 Introduction

1.1 Context: The Mural Proof Assistant

The problem addressed arose in the development of the Mural system.[1, 8] Mural is an interactive mathematical reasoning environment designed to assist the kind of theorem proving tasks that arise when following a formal methods approach to software engineering. It is the result of work carried out at Manchester University and the Rutherford Appleton Laboratory under the Alvey IPSE 2.5 project.

The Mural proof assistant is highly interactive in nature and considerable emphasis has been placed on the design of the user interface in an attempt to enable users to maintain their intuition of the problem domain and hence guide the proof in the right direction, whilst the accurate symbolic manipulation of the machine maintains the integrity of the proof.

Mural is founded on a generic logic and can be instantiated for reasoning in a variety of object logics. Logical theories are constructed in a hierarchical store where collections of declarations and axioms are structured along with derived rules and their proofs. Rules can have

any number of sequent or ordinary hypotheses and a single, ordinary conclusion. The syntax of the logic in question is defined in the Mural "meta-syntax". To give a few examples, there are Mural expressions forms for *compound expressions* like $f(x, y, z)$, *binder expressions*, like $\forall x{:}\, A \cdot P(x)$, and *compound types* like $A \times B$, and *subtypes* like $< x{:}\, T \mid P[x] >$.

Proofs are constructed in a natural deduction style, where the reasoning can be forwards, backwards or "mixed". That is proofs can be constructed from the hypotheses or the conclusion, or from some point in the middle. The choice of rules to apply is determined by the user although there is a tactics language that can be used to "code up" common reasoning strategies. As far as possible a *laissez-faire* attitude is taken to the construction of proofs – there is no insistence that rules must themselves be proven before they are used in other proofs – Mural carries out the book-keeping required in order to maintain integrity of the system, tracking the dependencies of proofs upon others and checking for circularities.

1.1.1 Pattern matching in Mural

In the normal way, inference rules of the object logic are expressed as axiom-schemata by augmenting the object level syntax with the notion of meta-variables that can be instantiated to any object level expression. The need for a matching algorithm arises when considering the use of an inference rule to construct a justification for a line in a proof. For example, a rule such as

$$\boxed{\text{=-subst}}\ \frac{a = b, P(a)}{P(b)}$$

can be applied to the lines

(1) $3 * x = f(x)$
(2) $even(3 * x)$

to justify the line

(3) $even(f(x))$

Alternatively this could be applied with line (1) as a hypothesis, line (3) as the conclusion, to derive line (2) as a new line to be justified.

In this case P is a "higher order" meta-variable of arity 1: its parameter a is itself a meta-variable of arity 0 which can be instantiated to any part of the matched expression for P. The potential matches are constrained by the same matching substitution being valid across all the expressions to be matched. Thus in the above simple example, the following substitution is used.

$a \mapsto 3 * x$
$b \mapsto f(x)$
$P \mapsto even(\_)$

Here, the underscore, "_", represents a space-holder, or "hole" in the pattern, which is filled in the generation of the object level expression. In practice these holes are numbered. Note that x is an object level variable, and as such is not instantiated in the matching process.

Thus the higher order nature of the matching refers to the possibility of having patterns within patterns, rather than a full β-reducing matching in the sense of Huet [6]. This makes the algorithm more straightforward and finitary. However, this syntax is still quite expressive. For example, higher-order meta-variables (those of non-nullary arity) can be nested. Thus, the expression $P[Q[a]]$ where a is an nullary meta-variable and P and Q are both unary higher order meta-variables can be matched to the term $3 + 4 + 5$ via the instantiation:

$$P \mapsto 3 + \_,\ Q \mapsto 4 + \_,\ x \mapsto 5.$$

The algorithm produces all the possible instantiations of meta-variables in the rule that lead to a match with the proof line. In the last example, the following three instantiations would also be valid:

$$P \mapsto 3 + 4 + 5,$$
$$P \mapsto 3 + 4 + \_,\ Q \mapsto 5,$$
$$P \mapsto 3 + 4 + \_,\ Q \mapsto \_,\ x \mapsto 5.$$

In total there are 32 valid solutions to this example.

The Mural system itself was initially specified in VDM [7]. The original algorithm was then coded directly from this high level specification. This was an exhaustive search method which sought out all the possible instantiations for each of the variables independently and then retrospectively analysed them to find out which combinations of the individual solutions had no clashes. While this was a succinct way of specifying the problem, as an implementation method it proved to be too inefficient to be practical. The central idea behind the algorithm presented here is straight forward. We must traverse the structural tree of the matched term building up tentative instantiations "as we go" and carrying them around with us. When a meta-variable is encountered the possible instantiations of it are compared to each of the partial instantiations we have so far; when clashes occur those candidates are rejected immediately. Whilst no claim is made that the algorithm presented here is in any way optimal, it does to some extent make use of available information and hence avoid redundant computation and in practise is sufficiently efficient for the interactive use for which it is intended.

The purpose of this paper is to exhibit the techniques employed in the deconstruction of the problem rather than to introduce any new or deep methods. Thus, the algorithm is exhibited over a very simple expression syntax. Binary trees suffice for our purposes.

1.2 Related Work

The problem of higher order unification is well documented, in particular by [6], where an algorithm is presented that generates unifiers in typed lambda calculus. This is the basis of the unification that drives the reasoning in Isabelle [10]. The problem considered here is more restricted in two ways. Firstly, following in the LCF style [4], meta-variables from the statement of the rule become parameters of the proof and are treated as though they were constants in the text of the proof, not amenable to instantiation, thus making the matching one-way. Secondly β-reduction is not primitive to the equivalence of expressions in our treatment. The problem of higher order matching with β-reduction has been less widely

researched. [2] gives a formal derivation of an algorithm for this in a restricted version of typed lambda calculus using transformational techniques. The distinction between this and our approach can be seen by considering the following example. Matching modulo β-reduction of a constant, E, against the higher order functional $X[\lambda x.x]$ yields the infinite set of solutions:

$$X \mapsto \lambda y.E,$$
$$X \mapsto \lambda y.y(E),$$
$$X \mapsto \lambda y.y(y(E)),$$
$$\ldots$$

Our approach would not permit these solutions since we insist that resulting instantiated forms must be syntactically identical rather than β-equivalent. This restriction ensures that there can only be finitely many matches.

We make no claim that the algorithm we present is the most efficient. For example, there is no attempt to exploit common information in the style of Martelli and Montenari [9]. Our treatment merely avoids some redundant recomputation by making use of the partial solutions already computed.

1.3 Overview

In the remainder of this section we give a brief review of some related work on higher-order matching. In Section 2 we give an informal description of the specific problem that we are tackling, and of how the problem was broken down. Section 3 introduces the notational conventions that we use and in the following sections, 4, 5 and 6, we give the specifications and implementations of the three cases of the problem in increasing generality. Section 7 gives a brief description of how this was then incorporated into Mural, and draws a few conclusions from the exercise.

2 Developing the algorithm

2.1 Extracting the essence of the problem

In order to address the development of an algorithm for the matching, the problem is subjected to a series of simplifications. Firstly, we only consider the matching of expressions; the generalisation to sequents and rules is not difficult. Secondly, we choose an object level syntax that is as simple as possible to demonstrate the issue in hand, labelled binary trees are our example. Again the generalisation, to the generic logic syntax of Mural, was straightforward.

The problem of matching considered in this paper is, given an expression without meta-variables, known as the *Matched Expression* and a sequence of expressions with meta-variables, known as the *Matching Expressions*, derive all the possible combinations of a *context*, that is, an expression with integer labelled *holes* for place-holders, and a substitution, in such a way that when each hole in the context is filled with the matching expression from

the sequence corresponding to the label of that hole, and then the substitution is applied, the matched expression results. We will give a formal description of this later. For now we present a small example. If the matched term is

$$a(1, b(2, 3)),$$

and the sequence of matching terms is

$$[x, b(2, y)],$$

with x and y meta-variables, then one possible solution is the context

$$a(\boxed{1}, \boxed{2}),$$

together with the substitution

$$x \mapsto 1,\ y \mapsto 3$$

where the $\boxed{\text{i}}$ are the labelled holes.

We break down the development of the algorithm into three stages each addressing successively more complex kind of matching. Firstly we just consider the framework for the matching by not allowing any meta-variables in the syntax. The only matches here are identities. Whilst this is a trivial step as there are no substitutions to be concerned with, it does set up the framework for the algorithm and establishes the flow of control to be employed. Secondly we introduce the possibility of meta-variables into the matching terms. These can match any expression in the matching terms, but do not allow these meta-variables themselves to have arguments. In this stage we introduce the matching substitution, and use the well-formedness of substitutions to determine the validity of a match while the pass over the matched term is still in progress. Lastly we extend to higher-order meta-variables. That is meta-variables with arguments which may themselves contain higher-order meta-variables. We will see later that, although it is this final generalisation that makes the problem interesting, it turns out to be a very simple matter to generalise the algorithm.

At each one of these stages the matching task is subdivided into three successively specialised functions. The most specific is the one that actually does matching. The others serve to provide the search over the matched expression and the sequence of matching expressions respectively. Thus, each stage of generalisation is basically a matter of adding extra cases to the matching function whilst the other two functions change little. In particular the final "interesting" generalisation is achieved simply by adding a function application that makes the three functions mutually recursive.

3 Notation

The specification language used is VDM [7] and the implementation language Standard-ML [5]. It is worth making a few short comments about these languages.

3.1 General style

We choose to specify our problem in VDM. Although we require only a restricted part of the language, the availability of subtyping (inclusion polymorphism) was of great use to us. The VDM specification language is based on set theory and a three valued logic of partial functions, LPF. Normally used to specify imperative programs, VDM incorporates a notion of state and operations on that state. However we only define types and functions on those types. Indeed we only make use of explicit (though non-constructive) functions.

Our implementation is in Standard ML. Standard ML is a strict, polymorphic functional programming language, though its type polymorphism does not allow for subtyping in the style of VDM, either by inclusion-polymorphism or by data-type invariants. The features of the language exploited here are those of user defined datatypes, pattern matching, and the use of simple higher order functions. Together these allow a great degree of brevity in the code and thus it proved to be an ideal language for experimentation with these algorithms, the brevity of the code exposing the control structure of the algorithm.

The two notations complemented each other very successfully and although no systematic proofs have been attempted, the combination of the complementary features of the two notations gives us confidence in the correctness of the implementation.

A degree of familiarity with both VDM and Standard ML (or at least functional programming) is assumed throughout this paper. Specialised constructs are explained, but the general methodology of both languages is not.

3.2 Specific conventions

The following notational conventions will be used throughout the formal texts that follow:

e, e_i, E, E_i will stand for expressions; where appropriate small letters will stand for the matching expression and capitals for the matched expression.

c, c_i for contexts.

S, S_i, S' for substitutions.

The following abbreviations will also be used for the abstract syntax of expressions, described in more detail below.

λ_i for `leaf(i)`,

$\eta_a(c_1, c_2)$ for `node("a",`c_1`,`c_2`)`,

$v_a[e_1, \ldots, e_n]$ for `var("a",`$[e_1, \ldots, e_n]$`)`, and

φ_i for `hole(i)`.

4 Spec 1: No Variables

In the first specification, only the framework for matching is set up. No instantiation of variables is considered - the only matches being identities. This is a trivial step as far as the matching is concerned, but it sets up the search strategy.

4.1 Abstract Syntax

Expressions are simply binary trees with leafs labelled by integers and nodes labelled by strings. We express this in a "ML like" syntax. Note that we cannot use this in the ML implementation as ML does not have subtyping. However, we wish to use subtyping in the VDM specification.

```
Exp = leaf of int
    | node of (string * Exp * Exp)
```

Contexts are expressions where any subtree can be replaced by a numbered hole. They are numbered so that they can be filled in systematically.

```
Context = leaf of int
        | hole of int
        | node of (string * Context * Context)
```

4.2 Function Specifications

We utilise an auxiliary function that describes the "filling-in" of holes in contexts:

$$\mathit{fillholes} : \mathit{Cont} \times \mathit{Exp}^* \to \mathit{Exp}$$

$$\mathit{fillholes}(c, [e_1, \ldots, e_n]) \triangleq c\,[e_i/\varphi_i]_{i=1,\ldots,\mathit{arity}(c)}$$

pre $\mathit{arity}(c) \leq n$

The arity of a context is the largest index of one of its holes.

We will write *fillholes* as an "infix symbol of zero width", i.e.

$$c[e_1, \ldots, e_n] \triangleq \mathit{fillholes}(c, [e_1, \ldots, e_n])$$

We define three matching functions that deal with decreasing levels of generality. Firstly *match* which finds all the possible matches:

$$\mathit{match} : \mathit{Exp} \times \mathit{Exp}^* \to \mathit{Cont}\text{-set}$$

$$\mathit{match}(E, [e_1, \ldots, e_n]) \triangleq \{c \mid c[e_1, \ldots, e_n] = E\}$$

For example:

$$match(\eta_a(\lambda_1, \lambda_2), [\lambda_1, \lambda_2, \eta_a(\lambda_1, \lambda_2)]) = [\eta_a(\lambda_1, \lambda_2), \eta_a(\lambda_1, \varphi_2), \eta_a(\varphi_1, \lambda_2), \eta_a(\varphi_1, \varphi_2), \varphi_3]$$

Secondly we define a function *herematch* that is very much like *match*, but additionally insists that the returned context is a *hole*. The specification can be written by adding a conjunct to the post condition:

$$herematch : Exp \times Exp^* \rightarrow Cont\text{-set}$$

$$herematch(E, [e_1, \ldots, e_n]) \triangleq \{c \mid c[e_1, \ldots, e_n] = E \wedge \exists j \in \{1, \ldots, n\} \cdot c = \varphi_j\}$$

However this specification simplifies to:

$$herematch : Exp \times Exp^* \rightarrow Cont\text{-set}$$

$$herematch(E, [e_1, \ldots, e_n]) \triangleq \{(c) \mid \exists j \in \{1, \ldots, n\} \cdot c = \varphi_j \wedge e_j = E\}$$

Since we know that the returned contexts are just indexed holes, alternatively, *herematch* could return a set of the indices.

Thirdly *fullmatch* has the further restriction that it just takes a single expression, at this level that makes it just a test for equality, later it will return substitutions.

$$fullmatch : Exp \times Exp \rightarrow \mathbb{B}$$

$$fullmatch(E, e) \triangleq e = E$$

4.3 Implementation

We give an implementation of the functions of *match*, *herematch* and *fullmatch*. Note that there is no subtyping in Standard ML and so the most general form for the abstract syntax is taken. Thus the datatype `Context` is used in all cases: the typing information given in ML comment brackets `(*...*)` is just an aid to the reader. Note that if a match against an ill-formed term (that is a hole) is attempted, then an the function `failwith` is called which raises an exception with a message. Also, where sets are used in the specification, lists are used in the implementation: though an abstract data type of sets could be defined we feel this is not relevant to the topic under consideration..

As already mentioned, a degree of familiarity with a functional programming style is assumed here. For the Standard-ML syntax, it is sufficient to note that function definitions are introduced by the keyword `fun`, local declarations within the keywords `let ...in ...end`, and case expressions with `case ...of ...=> ... | ...=> ....` Extensive use is made of pattern matching on the data type in function declarations and case expressions.

Also note that `_::_` is the infix constructor for concatenating onto the front of a list, `_@_` is the infix operator for appending two lists together, and `_o_` is the infix operator for function composition, $(f \circ g)x = f(g(x))$.

Now the three main functions.

```
(*
match : Exp -> Exp list -> Context list
*)
fun match (leaf i) el = leaf i :: herematch (leaf i) el
  | match (node(s,e1,e2)) el =
      let fun makenodes (l::ll) r = node(s,l,r)::makenodes ll r
            | makenodes [] _ = []
      in  herematch (node(s,e1,e2)) el
            @ mapapp (makenodes (match e1 el)) (match e2 el)
      end
  | match (hole _ ) _ = failwith "cannot match on a hole"
```

match is the top level function which controls the search. The crucial case is the second, where a call to herematch finds all the exact matches of this term with the elements of the expression list and then recursively calls itself on first the left and then the right subtrees to find all the possible matches on subterms. The local function makenodes rebuilds the expression with the holes in the subterms. mapapp is an auxiliary higher-order function.

```
(*
mapapp : ('a -> 'b list) -> 'a list -> 'b list
*)
fun mapapp(f,[]) = []
  | mapapp(f,h::tl) = f h @ mapapp(f,tl)
```

which takes a function returning a list, *maps* that function over a list and returns the resulting list formed by *appending* the resulting lists together.

```
(*
herematch : Exp -> Exp list -> Context list
*)
fun herematch e1 es =
    let fun herematchi (e::el) i =
                if fullmatch e1 e
                then hole i :: herematchi el (i+1)
                else herematchi el (i+1)
          | herematchi [] _ = []
    in herematchi es 1
    end;
```

herematch, which controls the matching of an entire expression "here", calls the fully-matching function with respect to each of the matching expressions in turn. At this, variable-free, level, the fully-matching function merely returns a boolean value. If there is a match of the matched expression's subexpression and the matching expression, the expression is replaced with a hole labelled by the number of the matching expression, given by its position in the list. Thus the ordering property of the list of matching expressions is vitally important if the result of the matching algorithm is going to be meaningful.

```
(*
```

```
fullmatch : Exp -> Exp -> bool
*)
fun fullmatch T1 T2 =
      (case (T1,T2) of
        (hole _ ,  _)       => failwith "cannot match on a hole"
      | (_ , hole _ )       => failwith "cannot match on a hole"
      | (leaf i,leaf j)     => i = j
      | (node(s,t1,t2),node(s',t1',t2')) =>
        s = s' andalso fullmatch t2 t2' andalso fullmatch t1 t1'
      | _                   => false )
      ;
```

`fullmatch` is the function which carries out the case analysis on the matched and matching expressions. At the variable free level this is a trivial inductive identity comparison of the labels on leafs and on nodes.

It can be seen that the exercise of matching between expressions which do not include variables is a comparatively trivial one. However it is one worth going through as it establishes several important points of strategy which become more important later.

5 Spec 2: Ordinary Variables

We now consider the extension of the matching algorithm to cater for the use of ordinary, first-order variables within the matching expressions.

5.1 Abstract Syntax

The expressions from above become the ground terms (full expressions) at this level:

```
FullExp = leaf of int | node of (string * FullExp * FullExp)
```

Normal expressions can have variables (named by strings) that will be instantiated by `FullExps`:

```
Exp = leaf of int
    | node of (string * Exp * Exp)
    | var of string
```

Contexts become:

```
Context = leaf of int
        | node of (string * Context * Context)
        | var of string
        | hole of int
```

We shall also sometimes refer to the type of *Variables*, which are Expressions restricted to variables, and use the obvious function that returns the variables in a context.

```
Variable =  var of string
```

$vars$: $Context \rightarrow Variable^*$

5.2 Substitutions

A substitution records an instantiation of variables. It is given as a finite function, or map in VDM terminology, from variables to expressions.

$$Substitution = Variable \xrightarrow{m} Exp$$

We distinguish substitution *Empty* which has the empty set as its domain.

$$\mathsf{dom}\ Empty = \{\}$$

The application of substitutions will be written like ordinary function application.

$$S(e) \triangleq e\,[e_i/v_i]_{(v_i \mapsto e_i) \in S}$$

We define two functions on substitutions that although not required in the specification will be used in the implementation. Firstly, a function that extends a substitution by adding a Variable/Expression pair in such a way that does not break the condition of it being a map.

$$add\_subst : Variable \times Expression \times Substitution \rightarrow Substitution$$

$$add\_subst(v, e, S) \triangleq S \dagger (v \mapsto e)$$

$$\mathsf{pre}\ v \in \mathsf{dom}\, S \Rightarrow S(v) = e$$

Secondly, the extension of this function which acts over a sequence of substitutions. For each substitution in the sequence, it attempts to extend it with the Variable/Expression pair. If it can do so it does, otherwise it discards the substitution as it would give a clash of assignments to the variable.

$$map\_add\_subst : Variable \times Expression \times Substitution^* \rightarrow Substitution\text{-}\mathsf{set}$$

$$map\_add\_subst(v, e, [S_1, \ldots, S_n]) \triangleq \{S' \mid \exists i \in \{1, \ldots, n\} \cdot S' = add\_subst(v, e, S_i)\}$$

Note that if the last conjunct in the predicate part of the expression holds then it must certainly be defined and hence its precondition must hold.

5.3 Function Specifications

Now we present the three main functions:

$$match : FullExp \times Exp^* \times Sub^* \rightarrow (Cont \times Sub)\text{-set}$$

$$match(E, [e_1, \ldots, e_n], [S_1, \ldots, S_k]) \triangleq$$
$$\{(c, S') \mid S'(c[e_1, \ldots, e_n]) = E \wedge$$
$$\exists i \in \{1, \ldots, k\} \cdot vars(c[e_1, \ldots, e_n]) \lhd S' \subseteq S_i \subseteq S'\}^1$$

The first conjunct says that the resulting substitution has the correct effect on the "filled" context. The second says that this new substitution extends one of those passed in and furthermore that no superfluous maplets are added to the substitution.

The next function adds the restriction that the returned context must be a hole:

$$herematch : FullExp \times Exp^* \times Sub^* \rightarrow (Cont \times Sub)\text{-set}$$

$$herematch(E, [e_1, \ldots, e_n], [S_1, \ldots, S_k]) \triangleq$$
$$\{(c, S') \mid S'(c[e_1, \ldots, e_n]) = E \wedge$$
$$\exists j \in \{1, \ldots, n\} \cdot c = \varphi_j \wedge$$
$$\exists i \in \{1, \ldots, k\} \cdot vars(c[e_1, \ldots, e_n]) \lhd S' \subseteq S_i \subseteq S'\}$$

Which simplifies to:

$$herematch : FullExp \times Exp^* \times Sub^* \rightarrow (Cont \times Sub)\text{-set}$$

$$herematch(E, [e_1, \ldots, e_n], [S_1, \ldots, S_k]) \triangleq$$
$$\{(c, S') \mid \exists j \in \{1, \ldots, n\} \cdot c = \varphi_j \wedge S'(e_j) = E \wedge$$
$$\exists i \in \{1, \ldots, k\} \cdot vars(c[e_1, \ldots, e_n]) \lhd S' \subseteq S_i \subseteq S'\}$$

The third function restricts further: by only taking a single expression as second parameter, there is no need to return a context, as the only one we are interested in would be φ_1.

$$fullmatch : FullExp \times Exp \times Sub^* \rightarrow Sub\text{-set}$$

$$fullmatch(E, e_1, [S_1, .., S_n]) \triangleq$$
$$\{S' \mid S'(e_1) = E \wedge$$
$$\exists i \in \{1, \ldots, k\} \cdot vars(c[e_1, \ldots, e_n]) \lhd S' \subseteq S_i \subseteq S'\}$$

5.4 Implementation

The major difference in the implementation of this second level of matching is the introduction of substitutions. This is represented as an abstract datatype.

```
abstype Substitution = Sub of (string * Context) list | Fail
```

Thus a substitution is represented by an association list of variables (represented by the string labelling them) and contexts. In addition, we introduce the element `Fail` which handles the exceptional case when in attempting to extend a substitution, a clash of variables is found and no valid substitution can be found. This case could be handled in Standard ML by

[1] $\lhd$ is relation domain deletion, defined by $S \lhd R = \{(a, b) \in R \mid a \notin S\}$

raising an exception which is handled by the calling function. However, it is much more within the functional programming style to lift the datatype with a error value.

We do not give the whole definition of the functions on substitution, merely the signatures of the supplied functions.

```
abstype Substitution
  val Empty = - : Substitution
  val add_subst = fn:(string * Context)->(Substitution->Substitution)
  val map_add_subst = fn : (string * Context) ->
                          (Substitution list -> Substitution list)
  val apply = fn : Context -> (Substitution -> Context)
  val printsub = fn : Substitution -> string
```

`Empty` is the identity substitution which assigns no variables.

`add_subst` inserts a variable (simply represented by `string`) and its assigned expression into an existing substitution. Should the variable already be assigned to another value, the `Fail` substitution is generated.

`map_add_subst` inserts a variable/context pair into every element of a list of substitutions. Should the failure substitution be generated at any stage it is immediately discarded. Thus by using this function, variable clashes are detected straightaway, and the substitutions are discarded.

Not all of these functions are used in the matching code: for example nowhere in the generation of the matching substitution do we actually apply a substitution. However, application can be used for checking the correctness of solutions found and so is included in the `abstype`.

With the introduction of substitutions, the three functions become:

```
(*
match : Exp -> Exp list -> Substitution list
                -> (Context * Substitution) list
*)
fun match (leaf i) Explist S =
        (map (pair (leaf i)) S) @ (herematch (leaf i) Explist S)
  | match (node(s,e1,e2)) Explist S =
     let fun makenodes (l,((r,m)::l')) =
                              (node(s,l,r),m)::makenodes (l,l')
           | makenodes (l,[]) = []
         val allmatches =
              mapapp (makenodes o
                   (applysndpair (match e2 Explist o singleton)))
                      (match e1 Explist S)
     in
       allmatches @ (herematch (node(s,e1,e2)) Explist S)
     end
  | match (hole _ ) TL S = failwith "cannot match on a hole"
  | match (var _) TL S   = failwith "cannot match on a var"
```

match is essentially undertaking the same strategy as before. On the matched expression being a node, in calculating the local value allmatches it finds all the matches on the left sub-expression, resulting in a list of context, substitution pairs and then matches on the right subexpressions in the environment of the each of the generated left subexpressions substitutions in turn. Thus any clashes between matches on the left and right subtrees are detected and eliminated as soon as they are found. makenodes rebuilds the contexts by taking the left hand context and the list of valid right-hand contexts and substitutions and reconstructing them returning a list of contexts at the level of the calling match.

applysndpair and singleton are two technical auxiliary functions:

```
(*
applysndpair : ('b -> 'c) -> ('a * 'b) -> ('a * 'c)
*)
fun applysndpair f (a,b) = (a,f b)
(*
singleton : 'a -> 'a list
*)
fun singleton a = [a];
```

the first takes a function and a pair, returning a new pair with the second element replaced by the result of applying the function to it, and the second takes any value and returns the singleton list of that value.

herematch is identical to that in the previous section except that it passes a substitution list to the fully matching function which returns a list of valid substitutions (rather than a boolean). These are then paired off with the hole labelled by the current matching expression.

```
(*
herematch : Exp -> Exp list -> Substitution list
                  -> (Context * Substitution) list
*)
fun herematch el es S =
    let fun herematchi (e::el) i =
            map (pair (hole i)) (fullmatch el e S)  @
                    herematchi el (i+1)
          | herematchi [] _ = []
    in herematchi es 1
    end
```

Again the function fullmatch is a case analysis which inductively checks for identities on the labels of leafs and node. However, when a matching variable is found, a new substitution is generated by assigning the matched expression to it. This is generated in the context of the current set (represented as a list) of substitutions, discarding (by the use of the map_add_subst function) those which would have variable clashes.

```
(*
fullmatch : Exp -> Exp -> Substitution list -> Substitution list
```

```
*)
fun fullmatch E1 E2 S =
      (case (E1,E2) of
        (hole _ , _)          => failwith "cannot match on a hole"
      | (_ , hole _)          => failwith "cannot match on a hole"
      | (var  _ , _)          => failwith "cannot match on a var"
      | (leaf i,leaf j)       => if i = j then S else []
      | (node(s,e1,e2),node(s',e1',e2'))  =>
        if s=s' then fullmatch e2 e2' (fullmatch e1 e1' S) else []
      | (_, var x)            => map_add_subst (x,E1) S
      | _                     => [] )
      ;
```

6 Spec 3: HO Variables

Finally we allows expressions with higher order variables. A higher order variable has as parameter a list of expressions, which can in themselves contain higher-order variable. Higher order variables can be instantiated to contexts and then the parameter expressions can fill any holes these contexts. When the list is empty they behave like ordinary variables.

6.1 Abstract Syntax

Full expressions are unchanged:

```
FullExp = leaf of int | node of (string * Exp * Exp)
```

Variables are extended to admit higher order variables:

```
Exp = leaf of int
    | node of (string * Exp * Exp)
    | var of (string * Exp list)
```

Contexts are expressions where any subtree can be replaced by a hole.

```
Context = leaf of int
        | node of (string * Context * Context)
        | var of (string  * Context list)
        | hole of int
```

Note variables in Contexts can have holes in their parameters but variables in Expressions cannot.

6.2 Substitutions

Substitutions must now allow for higher-order variables to be mapped to contexts which themselves may have holes within them. To cater for this, notion of the application of

substitutions has to be extended to allow for the substitution of parameters into these holes. Also, in the expression the substitution is being applied to, the variable is supplied with parameters, which themselves may have higher-order variables. These are used to fill in the holes of the context and the substitution is applied to them recursively. Formally, substitution application becomes:

$$S(e) \triangleq e\,[c_i[S(e_1),\ldots,S(e_n)]/v_i[e_1,\ldots,e_n]]_{(v_i \mapsto c_i) \in S}$$

Implicitly, there is a well-formedness condition on this formula. The *fillholes* function can only be applied to the context if the number of parameters supplied is less than or equal to the maximum number of formal parameters the higher-order variable can take.

In the degenerate case of a higher-order variable with no parameters, this becomes the same as the first-order case described previously.

6.3 Function Specifications

With the recognition that the notion of substitution has been extended to allow for higher-order variables, the specifications of the matching functions remain unchanged.

6.4 Implementation

Having set up the framework for the matching algorithm, modifying it to allow for the generation of higher-order contexts is very simple. `match` and `herematch` are identical. The only change which we need to carry out to the code is in the `var` case of `fullmatch`. This now splits into two cases. When the list of context expressions carried by the variable is empty, the variable acts as a normal first-order variable, and the new substitution list is formed in the same fashion as before. However, if the list of context expressions is non-empty then we make a call to the `match` function thus making the three functions mutually recursive.

```
fun match ...
and herematch ...
(*
fullmatch : Exp -> Exp -> Substitution list -> Substitution list
*)
and fullmatch E1 E2 S =
      (case (E1,E2) of
        (hole _ , _)          => failwith "cannot match on a hole"
      | (_ , hole _)          => failwith "cannot match on a hole"
      | (var  _ , _)          => failwith "cannot match on a var"
      | (leaf i,leaf j)       => if i = j then S else []
      | (node(s,e1,e2),node(s',e1',e2'))  =>
        if s=s' then fullmatch e2 e2' (fullmatch e1 e1' S) else []
      | (_ , var (x,[]))      => map_add_subst (x,E1) S
      | (_ , var (X,es))      => add_HO_subst X ( match E1 es S )
      | _                     => [] )
```

```
;
```

The extra case calls `match` using the current subexpression as the matched expression and the context list as the matching expressions, in the environment of the current valid substitutions. This will generate a list of contexts and substitutions. It is the contexts which form the expressions to be assigned to the higher-order variable `X`. Consequently, the function `add_HO_subst` is needed which adds the assignment of `X` to each of the substitutions, checking whether any clashes of higher-order variables occurs.

```
(*
val add_HO_subst = fn : string -> ((Context * Substitution) list
                                   -> Substitution list)
*)
fun add_HO_subst X ((c,ms)::rc) =
            map_add_subst (X,c) [ms] @ add_HO_subst X rc
  | add_HO_subst _ [] = []
```

When a higher-order match is made, there is a set of valid assignments produced, each with their own associated substitution. `add_HO_subst` passes across that list and inserts the maplet formed by the higher-order variable and the new term, and inserts it into the associated substitution using the previously defined `map_add_subst` function, which checks to see if the higher-order variable has already been assigned to something different (which could happen). The function then returns a list of these modified substitutions, with the failures omitted.

7 Generalisation to the Mural Syntax

We have considered the implementation of this matching algorithm on binary trees. In the full Mural system, it is used on a much larger syntax. However, the algorithm generalises in the obvious manner for the generalised tree syntax; more cases are required for the pattern-matching.

One technical point which is worth mentioning here is that in Mural, we do not always wish to produce matches for all higher-order variables. In the case where we are proving general derived inference rules for the logic, rather than theorems of the logic, we wish the meta-variables to be parameters to the proof and remain uninstantiated. For example, if we define $\wedge$ using $\vee$ and $\neg$.

$$\boxed{\wedge\text{-def}}\ \frac{\neg(\neg A \vee \neg B)}{A \wedge B}$$

We may wish to derive the normal $\wedge$-introduction rule.

$$\boxed{\wedge\text{-intro}}\ \frac{A, B}{A \wedge B}$$

During the proof of this rule from the other we do not wish meta-variables A and B to be amenable to instantiation. We thus need to have a mechanism for freezing meta-variables in order to make them immune from the matching.

The matching algorithm was successfully extended to full Mural syntax and has been implemented in Smalltalk-80 [3][2]. The translation of ML to Smalltalk80 was straightforward. Higher-order functions can be simulated in Smalltalk80 by the use of passing blocks as messages between objects which can be executed on arrival.

Its incorporation removed an aggravating bottleneck in the systems performance. The pattern matching is now acceptably fast on the size of example that generally arises in the application area.

8 Conclusions

The methodology used for this exercise was not the classical "waterfall" model of specification, refinement and implementation. Rather, we experimented with implementations at the same time as we developed our understanding of the problem. Development of the final versions of the specification and implementation went hand in hand.

The analysis of the problem through successive simplifications proved to be of great help in deepening our understanding of it. By firstly considering only the simplest expressions that exhibited the problem and then considering successively restricted special cases of the matching, we found that the part of the implementation which we anticipated would be the most difficult, that of handling the higher-order case, proved to be very simple once the correct framework had been set up.

References

[1] J.C.Bicarregui and B Ritchie. *Providing Support for the Formal Development of Software*. Proceedings of the First International Conference on Systems Development Environments and Factories.1989. (ed. H. Weber) Pitman (1990).

[2] P. Burton. *Transformational Derivation of an Algorithm for Higher Order Matching* Department of Computer Science Report No. 484, Queen Mary and Westfield College,London (1990).

[3] A. Goldberg, D.Robson. *Smalltalk 80, the language and its implementation* Addison Wesley, 1985.

[4] M. Gordon, A. Milner, C.P.Wadsworth. *Edinburgh LCF* LNCS 78, Springer Verlag, 1979.

[5] R.Harper, R.Milner, M.Toftes. *The Definition of Standard ML Version 2*. University of Edinburgh, LFCS Report **ECS-LFCS-88-62** (1988).

[2] We acknowledge the work of Bob Fields of Manchester University who made this extension, produced the Smalltalk implementation and integrated it within the Mural system.

[6] G.P.Huet. *A Unification Algorithm for Typed λ-Calculus.* Theoretical Computer Science **1** (1975) 27-57.

[7] C.B.Jones. *Systematic Software Development Using VDM (2nd Edition).* Prentice Hall International (1990).

[8] C.B.Jones(ed) *Mural, a Formal Development Support System.* Submitted for publication in LNCS series, Springer Verlag.

[9] A.Martelli, U.Montenari *An Efficient Unification Algorithm.* ACM Transactions on Programming Languages and Systems, 4(2), 258-282, (1982).

[10] L.C.Paulson. *Natural Deduction as Higher-Order Resolution.* J. Logic Programming **1986:3** 237-258.