

Chapter 6

Matching Mathematical Expressions

6.1 Introduction

This chapter contains a description of an algorithm for matching the mathematical expressions used in CARE language. Matches are described in terms of instantiation of formal parameters (see Section 4.2).

In Section 6.2 a definition for the relation *matches* is given. Matches between a pattern and query are defined in terms of instantiations of the pattern; e.g. for terms:

$$\mid \text{ matches} : \mathbb{P}(Term \times Term \times Inst)$$

In this section, the equivalence being used is alpha-equivalence (i.e. where two expressions are equal up to renaming of bound variables). Since CARE program development is usually done for “concrete” applications containing no parameters, it will be assumed in this section that no parameters appear in the query.

Section 6.3 contains implementations for the function *match* for each of the different kind of CARE expression. For terms, *match* returns a (finite) set of *matches* between a query and pattern.

$$\mid \text{ match} : Term \times Term \rightarrow \mathbb{F} Inst$$

Section 6.4 contains a formal justification that the *match* functions are correct with respect to the *matches* relation. Also sketched is an informal proof of completeness.

In Section 6.5 the *matches* relation is redefined by replacing alpha-equivalence by AC-equivalence. For example, since the operator $+ : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is associative and commutative, the terms $2+1$ and $1+2$, which do not match under alpha-equivalence, can be matched under AC-equivalence. Next an implementation is given for the function $match_{AC}$, which returns the set of matches up to AC-equivalence (with respect to a set of AC operators), between a pattern and query.

$$\mid \quad match_{AC} : \mathbb{F} OpName \times Term \times Term \rightarrow \mathbb{F} Inst$$

In Sections 6.6 and 6.7 two techniques for narrowing the search space are explored. Firstly in Section 6.6, type-constrained matching is defined. This method narrows the search space by eliminating any matches between patterns and queries which have incompatible types, or any matches which lead to type conflicts within parameter instantiations. An algorithm for applying type-constrained matching is defined, which includes as input the type of any of the free variables in the pattern or query.

$$\mid \quad match_{TC} : Term \times Term \times (Var \rightarrow Set) \rightarrow \mathbb{F} Inst$$

The second method for narrowing the search space, described in Section 6.7, is referred to as interactive matching. This technique allows the user to have a degree of control over the flow of the search, allowing them to terminate a search early, or skip over particular patterns, by giving an appropriate response after each match is found.

$$\mid \quad match_{interact} : Term \times Term \times seq Response \rightarrow \mathbb{F} Inst$$

The final enhancement examined in Section 6.8 is two-way matching, or unification. Here the case where the query also contains parameters is considered, and so the original one-sided matching is no longer sufficient. A basic algorithm for performing two-way matching is described briefly, noting that the focus of this section is not the development of the algorithm, but rather the consideration of some of the issues relating to two-way matching.

6.2 Specification of matching algorithm

Recall that the expressions given in CARE are either terms, formulae or sets (see page 24), which can include parameters ranging over functions, relations and types. Expressions can be adapted by instantiating some or all of the formal parameters contained within (see Section 4.2 for formal definition of the *instantiate* function).

A pattern p *matches* a query q if there is an instantiation i such that the instantiated pattern is *alpha-equivalent* to the query. For terms matching is formally defined as follows:

$$\left| \begin{array}{l} \text{matches} : \mathbb{P}(\text{Term} \times \text{Term} \times \text{Inst}) \\ \hline \forall p, q : \text{Term}; i : \text{Inst} \bullet \\ \text{matches}(p, q, i) \Leftrightarrow \text{instantiate}(p, i) =_{\alpha} q \end{array} \right.$$

Example 6.1 Given the pattern $f(g(x), h)$ and a query $m(a(x, b), c)$, where f , g and h are parameters, m and a are functions, b and c are constants, and x is a variable, then the following instantiations satisfy the matches relation:

$$\begin{aligned} I_1 &= \{f(s, t) \rightsquigarrow m(s, t), g(s) \rightsquigarrow a(s, b), h \rightsquigarrow c\} \\ I_2 &= \{f(s, t) \rightsquigarrow m(s, c), g(s) \rightsquigarrow a(s, b)\} \end{aligned}$$

□

6.3 Basic algorithm for matching

This section describes an implementation of a matching algorithm which satisfies the top-level specification given in Section 6.2 above. Section 6.3.1 describes the algorithm informally for terms and formulae, while the rest of the section formalises the algorithms.

6.3.1 Informal description

The algorithm works by structural induction on the pattern.

Term matching The algorithm for matching terms follows the structure of the abstract syntax for terms given in Section 2.2.2. Note that while placeholders only occur in expressions used in instantiations, they are considered here for completeness.

case 1: $p = \text{var } x$, where x is a variable.

p matches only the term x with a trivial instantiation.

case 2: $p = \text{ph}(j)$, where j is a natural number > 0 .

p matches queries $q = \text{ph}(j)$ with a trivial instantiation.

case 3: $p = f(a_1, \dots, a_m)$, where f is a non-parametric function.

p matches queries of the form $f(b_1, \dots, b_m)$, where a_j matches b_j for each j . The set of matches can be formed by merging (where possible) the sets of instantiations formed by matching a_j against b_j for each j .

case 4: $p = f(a_1, \dots, a_m)$, where f is a parametric function.

The set of matches of p with an arbitrary term e is formed by considering all disjoint sets of subtrees of e for which each subtree e_{sub} in the set matches an a_j for some $j \in 1 \dots m$. The sets of instantiations formed by matching each e_{sub} in the disjoint set with the corresponding a_j are merged to form a set of instantiations. These are then merged with the instantiation $\{f(x_1, \dots, x_m) \rightsquigarrow e'\}$, where e' is formed by replacing each subtree by the corresponding placeholder.

The complete set of instantiations is formed by taking the union of instantiation sets for all disjoint sets of subtrees, and finally filtering out any instantiations which contain free variables.

Example 6.2 Given a pattern $p = f(g(x, y), h(x, y))$ where f , g and h are all parameters, together with a query $e = (x + y) * (y - x)$. Two (of a number of) possible ways of matching p against e are:

$$\begin{aligned} &\{f(a, b) \rightsquigarrow a * b, g(a, b) \rightsquigarrow a + b, h(a, b) \rightsquigarrow b - a\} \\ &\{f(a, b) \rightsquigarrow b * a, g(a, b) \rightsquigarrow b - a, h(a, b) \rightsquigarrow a + b\} \end{aligned}$$

The first match is generated by considering the following disjoint subtrees of e :

$$\{x + y, y - x\}$$

The first subtree $x + y$ is matched against the first argument of the function application f - i.e. $g(x, y)$ - and the second subtree $y - x$ is matched against the second argument of f - i.e. $h(x, y)$. Replacing the subtrees $x + y$ and $y - x$ with the placeholders $ph(1)$ and $ph(2)$ respectively, results in the instantiation $f \rightsquigarrow ph(1) * ph(2)$, which can be rewritten by replacing the numeric placeholders with symbolic placeholders as $f(a, b) \rightsquigarrow a * b$.

By following the same process it can be shown that $g(x, y)$ matches $x + y$ with instantiation $g(a, b) \rightsquigarrow a + b$. Similarly $h(x, y)$ matches $y - x$ with instantiation $h(a, b) \rightsquigarrow b - a$.

The second match that is listed above can be generated in much the same way. Like the first match, the subtrees $x + y$ and $y - x$ are considered, but this time the first subtree is matched against the second argument of f , and the second subtree is matched against the first argument of f . \square

Formula matching The algorithm for matching formulae is very similar to that given for matching terms. Briefly it works by considering the different kinds of patterns. Truth, negations, binary connectives and relations are matched by matching the corresponding sub-expressions. For parametric relations the same sort of strategy is applied as that for parametric functions described above. The noteworthy case is quantified formulae, when the signatures of the variable declaration part of the pattern and query are the same, but the variables names do not agree. To allow for matching up to alpha-equivalence, the bound variables in the pattern and query are renamed to common new variables not appearing in the body of either formula.

Example 6.3 Given a pattern p :

$$\forall x : X, y : Y \bullet P(x) \wedge Q(f(x, y), g(x, y))$$

where P , Q , f and g are parameters, together with a query e :

$$\forall s : X, t : Y \bullet s > 0 \wedge R(s + t, t/s)$$

The first step in the process for matching these two quantified formulae is to rename the bound variables in the pattern as follows, $x \mapsto u, y \mapsto w$, to give:

$$\forall u : X, v : Y \bullet P(u) \wedge Q(f(u, v), g(u, v))$$

Similarly the query is renamed by the map $s \mapsto u, t \mapsto w$ to give:

$$\forall u : X, v : Y \bullet u > 0 \wedge R(u + v, u/v).$$

Next the formulae in the bodies of the two quantified expressions are matched. In this case these are matched by matching the left and right conjuncts in the pattern with the corresponding conjuncts in the query. That is $P(u)$ is matched against $u > 0$ and $Q(f(u, v), g(u, v))$ against $R(u + w, w/u)$. The left conjuncts match with instantiation $P(a) \rightsquigarrow a > 0$. To generate matches for the right-hand conjuncts, a similar process to that described in Example 6.2 is followed.

Firstly consider the subtrees $u + v$ and v/u of the query, and match against the first and second arguments of the relation application Q . This results in the following instantiation:

$$\{Q(a, b) \rightsquigarrow R(a, b), f(a, b) \rightsquigarrow a + b, g(a, b) \rightsquigarrow b/a\}$$

By matching the same subtrees of the query as above, with the opposite arguments of Q , the following instantiation is generated:

$$\{Q(a, b) \rightsquigarrow R(b, a), f(a, b) \rightsquigarrow b/a, g(a, b) \rightsquigarrow a + b\}$$

□

The algorithm for matching mathematical sets is straightforward.

6.3.2 Preliminary definitions

Two instantiations are *mergeable* if they agree for common parameters. Instantiations are merged by combining the corresponding function, relation and set instantiation mappings using function override.

$$\begin{array}{l}
\hline
\text{areMergeableInsts} : \text{Inst} \times \text{Inst} \rightarrow \text{Inst} \\
\hline
\forall i, i_1, i_2 : \text{Inst} : \text{areMergeableInst}(i_1, i_2, i) \Leftrightarrow \\
\quad \forall f : \text{dom } i_1.\text{finsts} \cap \text{dom } i_2.\text{finsts} \bullet i_1.\text{finsts}(f) = i_2.\text{finsts}(f) \wedge \\
\quad \forall f : \text{dom } i_1.\text{rinsts} \cap \text{dom } i_2.\text{rinsts} \bullet i_1.\text{rinsts}(f) = i_2.\text{rinsts}(f) \wedge \\
\quad \forall f : \text{dom } i_1.\text{sinsts} \cap \text{dom } i_2.\text{sinsts} \bullet i_1.\text{sinsts}(f) = i_2.\text{sinsts}(f) \wedge \\
\quad i.\text{finsts} = i_1.\text{finsts} \oplus i_2.\text{finsts} \wedge \\
\quad i.\text{rinsts} = i_1.\text{rinsts} \oplus i_2.\text{rinsts} \wedge \\
\quad i.\text{sinsts} = i_1.\text{sinsts} \oplus i_2.\text{sinsts}
\end{array}$$

Example 6.4 Given the instantiations $i_1 = \{p_1 \rightsquigarrow e_1, p_2 \rightsquigarrow e_2, p_3 \rightsquigarrow e_3\}$, $i_2 = \{p_2 \rightsquigarrow e_2, p_4 \rightsquigarrow e_4\}$ and $i_3 = \{p_1 \rightsquigarrow e_5, p_2 \rightsquigarrow e_2, p_5 \rightsquigarrow e_5\}$, then i_1 and i_2 are mergeable giving the instantiation $\{p_1 \rightsquigarrow e_1, p_2 \rightsquigarrow e_2, p_3 \rightsquigarrow e_3, p_4 \rightsquigarrow e_4\}$. Similarly i_2 and i_3 are mergeable giving the instantiation, $\{p_1 \rightsquigarrow e_5, p_2 \rightsquigarrow e_2, p_4 \rightsquigarrow e_4\}$. The instantiations i_1 and i_3 are not mergeable, since they do not agree for the parameter p_1 . \square

A set of instantiations is mergeable if its elements are pairwise mergeable, i.e. any distinct pair of instantiations in the set are mergeable:

$$\begin{array}{l}
\hline
\text{mergeInstSet} : \mathbb{F} \text{Inst} \rightarrow \text{Inst} \\
\hline
\text{dom } \text{mergeInstSet} = \\
\quad \{is : \mathbb{F} \text{Inst} \mid \forall i_1, i_2 \in is \bullet (i_1, i_2) \in \text{dom } \text{areMergeableInsts}\} \\
\text{mergeInstSet}(\emptyset) = \text{trivInst} \\
\forall i : \text{Inst} \bullet \text{mergeInstSet}(\{i\}) = i \\
\forall is_1, is_2 : \mathbb{F} \text{Inst} \bullet \text{mergeInstSet}(is_1 \cup is_2) = \\
\quad \text{areMergeableInsts}(\text{mergeInstSet}(is_1), \text{mergeInstSet}(is_2))
\end{array}$$

Also defined is the trivial instantiation, which leaves any expression unchanged upon instantiation:

$$\begin{array}{l}
\hline
\text{trivInst} : \text{Inst} \\
\hline
\text{trivInst}.\text{finsts} = \emptyset \wedge \text{trivInst}.\text{rinsts} = \emptyset \wedge \text{trivInst}.\text{sinsts} = \emptyset
\end{array}$$

6.3.3 Terms

The function *match* is overloaded for the various kinds of expressions. For terms it takes two terms, the first being the pattern, the second the query, and returns a set of instantiations.

$$\mid \text{ match} : \text{Term} \times \text{Term} \rightarrow \mathbb{F} \text{Inst}$$

Alg. 1 contains an implementation of *match* for terms. It is implemented by considering the different kinds of patterns separately. For non-parametric functions a recursive call to the *match* function (in this case for lists of terms) is required, while for parametric functions a call to the function $\text{match}_{\text{fnct}}$, described later, is required.

Algorithm 1 Calculate $is = \text{match}(p, q)$ for terms

```

1: case  $p$  of
2:   var  $x$ :
3:     if  $q = \text{var } x$  then
4:        $is := \{\text{trivInst}\}$ 
5:   phj:
6:     if  $q = \text{phj}$  then
7:        $is := \{\text{trivInst}\}$ 
8:   fnApplic( $f, as_1$ ):
9:     if  $q = \text{fnApplic}(f, as_2)$  then
10:       $is := \text{match}(as_1, as_2)$ 
11:   termparam( $f, as$ ):
12:      $is := \text{match}_{\text{fnct}}(f, as, q)$ 

```

The *match* function for lists of terms is declared as follows:

$$\mid \text{ match} : \text{seq Term} \times \text{seq Term} \rightarrow \mathbb{F} \text{Inst}$$

The implementation of the *match* function for lists of terms is given in Alg. 2. It is implemented by considering empty and non-empty lists separately.

Parametric functions Matching of a parametric function pattern against an arbitrary query is modelled by the function $\text{match}_{\text{fnct}}$ (cf. the fourth case of the informal algorithm description in Section 6.3.1 above for more details).

$$\mid \text{ match}_{\text{fnct}} : \text{FunctionParam} \times \text{seq Term} \times \text{Term} \rightarrow \mathbb{F} \text{Inst}$$

Algorithm 2 Calculate $is = match(p, q)$ for lists of terms

```

1:  $is := \emptyset$ 
2: case  $p$  of
3:    $\langle \rangle$ :
4:     if  $q = \langle \rangle$  then
5:        $is := is \cup \{trivInst\}$ 
6:    $\langle h_1 \rangle \wedge t_1$ :
7:     if  $q = \langle h_2 \rangle \wedge t_2$  then
8:        $is_1 := match(h_1, h_2)$ 
9:        $is_2 := match(t_1, t_2)$ 
10:    for all  $i_1 \in is_1, i_2 \in is_2$  do
11:      if  $areMergeableInsts(i_1, i_2, i)$  then
12:         $is := is \cup \{i\}$ 

```

Before providing an implementation for the function $match_{funct}$, two auxiliary functions are defined and implemented. The function $replacement$ takes as inputs a term e corresponding to the query and a list of terms as , corresponding to the arguments of the parametric function application, both supplied as inputs to the $match_{funct}$ function. A set of pairs consisting of a $replacement$ term and an associated instantiation are returned.

$$| \quad replacement : Term \times seq\ Term \rightarrow \mathbb{F}(Term \times Inst)$$

The terms returned by the $replacement$ function represent ways in which the subterms e_{sub} of the query e can be replaced by placeholders of the form $ph\ j$, such that e_{sub} matches the j th term in the term list as . More precisely, for a term e , either:

1. leave e unchanged, i.e. replace no subterms; it is assumed in this case that e doesn't contain any placeholders,
2. replace e by a placeholder of the form $ph(j)$ where $j \in 1 \dots \#as$, or
3. if e is a function call, apply the replacement function recursively to the argu-

ments, and merge the results (where possible).

Example 6.5 Given the query $e = \text{cons}(\text{nil}, x)$, then possible replacements of e include the original query (leaving the query unchanged), the terms $\langle 1 \rangle$ and $\langle 2 \rangle$ (replacing the entire query by a placeholder) and the terms $\text{cons}(\langle 1 \rangle, x)$, $\text{cons}(\langle 1 \rangle, \langle 2 \rangle)$, etc. (replacing sub-expressions by placeholders).

Algorithm 3 Calculate $rs = \text{replacement}(e, es)$

```

1:  $rs := \emptyset$ 
2: if  $\neg \text{containsPlaceHolders}(e)$  then
3:    $rs := rs \cup \{(e, \text{trivInst})\}$ 
4: for all  $j \in \text{dom } es$  do
5:   for all  $i \in \text{match}(es(j), e)$  do
6:      $rs := rs \cup \{(\text{ph } j, i)\}$ 
7:   if  $e = \text{fnApplic}(f, as)$  then
8:     for all  $(as', i) \in \text{replacementList}(as, es)$  do
9:        $rs := rs \cup \{(\text{fnApplic}(f, as'), i)\}$ 

```

The implementation of *replacement* is given in Alg. 3. The third case given in this algorithm is implemented by calling the function *replacementList*.

| $\text{replacementList} : \text{seq Term} \times \text{seq Term} \rightarrow \mathbb{F}(\text{seq Term} \times \text{Inst})$

The function *replacementList* applies *replacement* recursively to an argument list (corresponding to the arguments of function call), and merges the resulting instantiations where possible (see Alg.4 for more details).

The function $\text{match}_{\text{fnct}}$ implements the fourth case given in the informal description of the matching algorithm for terms on page 122. $\text{match}_{\text{fnct}}$ is applied to a parametric function f , and its list of arguments es (which together represent the pattern), as well as an arbitrary term e (representing the query); the implementation is given in Alg. 5. The function $\text{replacement}(e, es)$ is called to calculate the set of term/instantiation pairs (e', i_1) , where e' represents a replacement of the subterms of e by placeholders and i_1 represents matches between the subterms and

Algorithm 4 Calculate $rs = replacementList(as, es)$

```

1:  $rs := \emptyset$ 
2: case  $as$  of
3:    $\langle \rangle$ :
4:      $rs := rs \cup \{(\langle \rangle, trivInst)\}$ 
5:    $\langle h_1 \rangle \wedge t_1$ :
6:      $rs_1 := replacement(h_1, es)$ 
7:      $rs_2 := replacementList(t_1, es)$ 
8:     for all  $(h_2, i_1) \in rs_1, (t_2, i_2) \in rs_2$  do
9:       if  $areMergeableInsts(i_1, i_2, i)$  then
10:         $rs := rs \cup \{(\langle h_2 \rangle \wedge t_2, i)\}$ 

```

terms in es , in accordance with the definition of *replacement*. The result is formed by merging the instantiation i_1 with the instantiation of f to e' for all possible pairs (e', i_1) .

Algorithm 5 Calculate $is = match_{funct}(f, es, e)$

```

1:  $is := \emptyset$ 
2: for all  $(e', i_1) \in replacement(e, es)$  such that  $freeVars(e') = \emptyset$  do
3:    $i_2.finsts := \{f \mapsto e'\}, i_2.rinsts := \emptyset, i_2.sinsts := \emptyset$ 
4:   if  $areMergeableInsts(i_1, i_2, i)$  then
5:      $is := is \cup \{i\}$ 

```

6.3.4 Formulae

The function *match* is also defined for formulae. It can be implemented by considering the different kinds of formulae in the pattern. The implementation is quite similar to the *match* function for terms. The pattern *true* will match only itself with the trivial instantiation. Not, binary connectives and relations are matched by matching the corresponding sub-expressions. Matching of parametric relations is very similar to matching of parametric functions. For quantified formulae, the auxiliary function $match_{quant}$, which is defined separately below, is called.

$$\mid \text{match} : Fmla \times Fmla \rightarrow \mathbb{F} Inst$$

Quantified formulae The auxiliary function $match_{quant}$ is called by the $match$ function for calculating the set of matches between two quantified formulae with the same quantifier. Recall that a quantified formula consists of a quantifier (such as universal or existential for example), a mathematical variable declaration list (cf. $MathVarDecls$ defined on page 28) , and a formula. The latter two are passed to the $match_{quant}$ function for both the pattern (given first) and the query.

$$\mid \text{match}_{quant} : MathVarDecls \times Fmla \times MathVarDecls \times Fmla \rightarrow \mathbb{F} Inst$$

Before describing the implementation of $match_{quant}$, two auxiliary functions are required. The first of these, $create_new_vars$, is used to create a list of variables of a fixed length, such that the any variables in the list do not appear in a predefined “forbidden” set of variables. The function takes as inputs a natural number n and a set of (forbidden) variables vs , and returns an injective sequence of variables. The resulting variable list has length n and its range is disjoint from the set vs .

$$\left| \begin{array}{l} \text{create_new_vars} : \mathbb{N} \times \mathbb{F} Var \rightarrow \text{iseq } Var \\ \hline \forall n : \mathbb{N}; vs : \mathbb{F} Var; vl : \text{iseq } Var \bullet vl \in \text{create_new_vars}(n, vs) \\ \Rightarrow \#vl = n \wedge \text{ran } vl \cap vs = \emptyset \end{array} \right.$$

Also defined is the function $match_{mvs}$ which returns the set of matches between two mathematical variable declaration lists.

$$\mid \text{match}_{mvs} : MathVarDecls \times MathVarDecls \rightarrow \mathbb{F} Inst$$

The function $match_{quant}$ is implemented in Alg. 6. It takes as inputs a variable declaration vs_1 and a formula f_1 from the pattern, and a variable declaration vs_2 and formula f_2 from the query. The first step is to match the variable declaration lists vs_1 and vs_2 to give a set of instantiations is_1 . Next a new list of variables vs is created, whose length is equal to the length of vs_1 and vs_2 , such that the variables in vs are distinct from the free variables of the pattern and query. In line 7 of the algorithm, any variables in f_1 bound by the variables in vs_1 are renamed to the variable in the corresponding position in the variable list vs . Similarly in line 8 of the algorithm

the variables in f_2 are renamed. These renamed formulae are then matched to give a second set of instantiations is_2 . The instantiation sets is_1 and is_2 are pairwise merged to give the result.

Algorithm 6 Calculate $is = match_{quant}(vs_1, f_1, vs_2, f_2)$

```

1:  $is := \emptyset$ 
2: if  $\#vs_1 = \#vs_2$  then
3:    $is_1 := match_{mvs}(vs_1, vs_2)$ 
4:    $n := \#vs_1$ 
5:    $fvars := (freeVars(f_1) \setminus varSet(vs_1)) \cup (freeVars(f_2) \setminus varSet(vs_2))$ 
6:    $vs := create\_new\_vars(n, fvars)$ 
7:    $f'_1 := renameVars(f_1, \{k : 1.. \#vs_1 \bullet \pi_1 vs_1(k) \mapsto vs(k)\})$ 
8:    $f'_2 := renameVars(f_2, \{k : 1.. \#vs_2 \bullet \pi_1 vs_2(k) \mapsto vs(k)\})$ 
9:    $is_2 := match(f'_1, f'_2)$ 
10:  for all  $i_1 \in is_1, i_2 \in is_2$  do
11:    if  $areMergeableInsts(i_1, i_2, i)$  then
12:       $is := is \cup \{i\}$ 

```

6.3.5 Sets

Algorithms for matching sets can be developed in a similar manner to the matching algorithms above. A given set matches only itself. Two constructed sets match if they have the same type constructor, and if their arguments match. A parametric set matches any other set.

$$\left| \begin{array}{l} match : Set \times Set \rightarrow \mathbb{F} Inst \\ match : seq Set \times seq Set \rightarrow \mathbb{F} Inst \end{array} \right.$$

6.4 Proof of correctness for match algorithms

The aim of this section is to prove that the algorithms for matching expressions given in Section 6.3 satisfy the relation $matches$ given in Section 6.2. This proof obligation is stated more precisely in the following theorem.

Theorem 6.1 *For all patterns p and queries q , if p matches q with instantiation i , then instantiating p with i yields q . i.e. $i \in \text{match}(p, q) \Rightarrow \text{matches}(p, q, i)$.*

The theorem shall be proved by considering the different kinds of patterns separately. The algorithms for matching variables, non-parametric functions, parametric functions, lists of terms and quantified formulae are the archetypical cases, so it is sufficient to prove these cases.

Note that since the matching algorithm is defined recursively, the theorem will be proved inductively, by assuming that the theorem holds for any recursive call to *match*.

The following shall be assumed in proving correctness.

Assumption 6.1 *For all patterns p , queries q and instantiations i*

$$i \in \text{match}(p, q) \Rightarrow (p, i) \in \text{dom instantiate}$$

6.4.1 Preliminaries

Before establishing the correctness of the matching algorithms given in the previous section, a number of properties are established.

Lemma 6.1 *Given any instantiations i_1 and i_2 which are mergeable (yielding the instantiation i), if an expression e can be instantiated by i_1 and i_1 is a complete instantiation of e , then e can also be instantiated by i , giving an equivalent result in both cases; i.e.*

$$\begin{aligned} (e, i_1) \in \text{dom instantiate} \wedge \text{areMergeableInsts}(i_1, i_2, i) \\ \Rightarrow (e, i) \in \text{dom instantiate} \wedge \text{instantiate}(e, i) =_{\alpha} \text{instantiate}(e, i_1) \end{aligned}$$

Proof: The above lemma follows from the fact that all parameters in h are instantiated by i_1 and that all parameters contained in i_1 are instantiated to equivalent values by i . \square

Lemma 6.2 *For instantiations i_1, i_2 and i*

$$\text{areMergeableInsts}(i_1, i_2, i) \Leftrightarrow \text{areMergeableInsts}(i_2, i_1, i)$$

Proof: This can be easily proven using the definition of *areMergeableInsts*. \square

The following corollary follows from lemmas 6.1 and 6.2.

Corollary 6.1 *Given instantiations i_1 and i_2 , provided i_2 is a complete instantiation of e , then*

$$\begin{aligned} (e, i_2) \in \text{dom instantiate} \wedge \text{areMergeableInsts}(i_1, i_2, i) \\ \Rightarrow (e, i) \in \text{dom instantiate} \wedge \text{instantiate}(e, i) =_{\alpha} \text{instantiate}(e, i_2) \end{aligned}$$

Lemma 6.3 *Given a parametric function application $\text{termparam}(f, as)$ and an instantiation i that maps f to a term e containing no placeholders, then $\text{termparam}(f, as)$ is instantiated to e by i . i.e.*

$$\begin{aligned} \neg \text{containsPlaceHolders}(e) \wedge f \mapsto e \in i.\text{finsts} \\ \Rightarrow \text{instantiate}(\text{termparam}(f, as), i) = e \end{aligned}$$

Proof. To prove the above lemma we firstly use the definition of *instantiate*, then make use of the fact that substituting placeholders in an expression e containing no placeholders will return e .

$$\begin{aligned} & \text{instantiate}(\text{termparam}(f, as), i) \\ &= \text{substPlaceHolders}(i.\text{finsts}(f), m) \quad [\text{Defn. of instantiate}] \\ &= \text{substPlaceHolders}(e, m) \\ &= e \quad \square \end{aligned}$$

Lemma 6.4 *Given terms e and e' , list of terms es and instantiation i such that*

$$(e', i) \in \text{replacement}(e, es),$$

for all $j \in \text{dom } es$ such that the placeholder $\text{ph}(j)$ is contained in the term e' , then $(es(j), i) \in \text{dom instantiate}$.

Proof. The above lemma can be proved by considering where the different results (e', i) arise in the implementation of *replacement* given in Alg. 3 on page 128.

1. $e' = e$, $i = \text{trivInst}$ and $\neg \text{containsPlaceHolders}(e)$. There are no placeholders in e , so there is nothing to check.

2. $j \in \text{dom } es$, $e' = \text{ph } j$ and $i \in \text{match}(es(j), e)$. This follows from Assumption 6.1.
3. $e' = \text{fnApplic}(f, as')$ and $e = \text{fnApplic}(f, as)$. This case can be proved by an inductive argument. \square

Two corollaries follow from the above lemma.

Corollary 6.2 *Given terms e and e' , a list of terms es and an instantiation i such that $(e', i_1) \in \text{replacement}(e, es)$ and $\text{areMergeableInsts}(i_1, i_2, i)$, then*

$$\begin{aligned} & \text{instantiate}(\text{substPlaceHolders}(e', es), i) \\ &=_{\alpha} \text{instantiate}(\text{substPlaceHolders}(e', es), i_1) \end{aligned}$$

Proof: The results follows simply from lemma 6.1 upon showing that:

$$(\text{substPlaceHolders}(e', es), i_1) \in \text{dom } \text{instantiate}.$$

However this follows from Lemma 6.4 by observing that term $\text{substPlaceHolders}(e', es)$, is equal to the term formed by replacing placeholders $\text{ph}(j)$ in e' by the terms $es(j)$. \square

Corollary 6.3 *Given lists of terms as_1 , as_2 and es and instantiations i_1, i_2 and i such that $(as_2, i_2) \in \text{replacementList}(as_1, es)$ and $\text{areMergeableInsts}(i_1, i_2, i)$ then:*

$$\begin{aligned} & \text{instantiate}(\text{substPlaceHolders}(as_2, es), i) \\ &=_{\alpha} \text{instantiate}(\text{substPlaceHolders}(as_2, es), i_2) \end{aligned}$$

Lemma 6.5 *Given terms e and e' , a sequence of terms es and an instantiation i , then*

$$\begin{aligned} & (e', i) \in \text{replacement}(e, es) \wedge \\ & \Rightarrow \text{instantiate}(\text{substPlaceHolders}(e', es), i) =_{\alpha} e \end{aligned}$$

Proof. Consider the three different cases in the implementation of the *replacement* function (see Alg. 3 on page 128).

1. $e' = e$, $i = \text{trivInst}$ and $\neg \text{containPlaceHolders}(e)$. Focusing on the left-hand side of the consequent:

$$\begin{aligned} & \text{instantiate}(\text{substPlaceHolders}(e', es), i) \\ &= \text{instantiate}(\text{substPlaceHolders}(e, es), \text{trivInst}) \\ &= \text{instantiate}(e, \text{trivInst}) = e \end{aligned}$$

2. $j \in \text{dom } es$, $i \in \text{match}(es(j), e)$ and $e' = \text{ph } j$. Using the definition of *substPlaceHolders* it follows that:

$$\begin{aligned} & \text{instantiate}(\text{substPlaceHolders}(e', es), i) \\ &= \text{instantiate}(\text{substPlaceHolders}(\text{ph } j, es), i) \\ &= \text{instantiate}(es(j), i) \\ &=_{\alpha} e \end{aligned}$$

3. $e = \text{fnapplic}(f, as)$, $e' = \text{fnapplic}(f, as')$ and $(as', i) \in \text{replacementList}(as, es)$.

This case can be proven by induction on as . If $as = \langle \rangle$, then it follows from line 4 of Alg. 4 that $as' = \langle \rangle$ and $i = \text{trivInst}$, therefore:

$$\begin{aligned} & \text{instantiate}(\text{substPlaceHolders}(e', es), i) \\ &= \text{instantiate}(\text{substPlaceHolders}(\text{fnApplic}(f, \langle \rangle), es), \text{trivInst}) \\ &= \text{substPlaceHolders}(\text{fnApplic}(f, \langle \rangle), es) \quad [\text{Defn. of instantiate}] \\ &= \text{fnApplic}(f, \langle \rangle) \quad [\text{Defn. of subst.}] \\ &= e \end{aligned}$$

Now consider $as = \langle h_1 \rangle \hat{\wedge} t_1$. From the implementation of *replacementList*, there exists expressions h_2 and t_2 , and instantiations i_1, i_2 such that $as' = \langle h_2 \rangle \hat{\wedge} t_2$, $(h_2, i_1) \in \text{replacement}(h_1, es)$, $(t_2, i_2) \in \text{replacementList}(t_1, es)$ and *areMergeableInsts*(i_1, i_2, i).

Focusing on the left-hand side of the consequent:

$$\begin{aligned} & \text{instantiate}(\text{substPlaceHolders}(e', es), i) \\ &= \text{instantiate}(\text{substPlaceHolders}(\text{fnapplic}(f, \langle h_2 \rangle \hat{\wedge} t_2), es), i) \\ &= \text{instantiate}(\text{fnapplic}(f, \langle \text{substPlaceHolders}(h_2, es) \rangle \\ & \quad \hat{\wedge} \text{substPlaceHolders}(t_2, es)), i) \quad [\text{Defn. of subst}] \\ &= \text{fnapplic}(f, \langle \text{instantiate}(\text{substPlaceHolders}(h_2, es), i) \rangle \\ & \quad \hat{\wedge} \text{instantiate}(\text{substPlaceHolders}(t_2, es), i)) \quad [\text{Defn. of instantiate}] \\ &=_{\alpha} \text{fnapplic}(f, \langle \text{instantiate}(\text{substPlaceHolders}(h_2, es), i_1) \rangle \\ & \quad \hat{\wedge} \text{instantiate}(\text{substPlaceHolders}(t_2, es), i_2)) \quad [\text{Cor.6.2 + Cor.6.3}] \\ &=_{\alpha} \text{fnapplic}(f, \langle h_1 \rangle \hat{\wedge} t_1) \quad [\text{Induction hypothesis}] \\ &= e \quad \square \end{aligned}$$

The proof of the following lemma is straightforward. It states that the result of replacing all placeholders $\text{ph } j$ by $\text{instantiate}(es(j), i)$ is the same as replacing the placeholders $\text{ph } j$ by $es(j)$ and then applying the instantiation i to the result.

Lemma 6.6 *Given a term e , a list of terms es and an instantiation i , then:*

$$\begin{aligned} & \text{substPlaceHolders}(e, \{j : 1 \dots \#es \bullet j \mapsto \text{instantiate}(es(j), i)\}) \\ & = \text{instantiate}(\text{substPlaceHolders}(e, es), i) \end{aligned}$$

The following lemma can be proven by using the definition of *instantiate* and considering the different kinds of expressions.

Lemma 6.7 *For all expressions e_1 , e_2 and instantiations i then $e_1 =_\alpha e_2$ implies $\text{instantiate}(e_1, i) =_\alpha \text{instantiate}(e_2, i)$*

6.4.2 Proof of correctness for terms

To prove Theorem 6.1, i.e. for all patterns p , queries q and instantiations i , $i \in \text{match}(p, q) \Rightarrow \text{matches}(p, q, i)$, the different kinds of expressions are considered. In this section, the theorem is proved for terms, lists of terms and quantified formulae. Since the match function is defined recursively, an inductive proof is required. The base cases for induction are the terminal nodes in the algorithm, i.e. variables, placeholders and empty term lists. The other cases - function applications, parametric function applications, lists of terms and quantified formulae - are implemented recursively. To prove these cases satisfy the theorem, the induction hypothesis will be assumed for recursive calls to the match function. As mentioned earlier, the proof for the other kinds of expressions is similar.

6.4.2.1 Variables

For a pattern of the form $p = \text{var } x$, it follows from Alg. 1 that $q = \text{var } x$ and $i = \text{trivInst}$. Therefore:

$$\begin{aligned} & \text{instantiate}(p, i) \\ & = \text{instantiate}(\text{var } x, \text{trivInst}) \\ & = \text{var } x = q \end{aligned}$$

6.4.2.2 Placeholders

For a pattern of the form $p = \text{ph}(j)$, it follows from Alg. 1 that $q = \text{ph}(j)$ and $i = \text{trivInst}$. Therefore:

$$\begin{aligned} & \text{instantiate}(p, i) \\ &= \text{instantiate}(\text{ph}(j), \text{trivInst}) \\ &= \text{ph}(j) = q \end{aligned}$$

6.4.2.3 Empty term lists

For an empty list $p = \langle \rangle$, it follows from Alg. 2 that $q = \langle \rangle$ and $i = \text{trivInst}$. Using this information it follows that:

$$\begin{aligned} & \text{instantiate}(p, i) \\ &= \text{instantiate}(\langle \rangle, \text{trivInst}) \\ &= \langle \rangle = q \end{aligned}$$

6.4.2.4 Non-empty lists

Consider a non-empty list of terms, $p = \langle h_1 \rangle \hat{\ } t_1$. From Alg. 2 for matching lists of terms, for each q and i satisfying $i \in \text{match}(p, q)$, there exists expressions h_2 , t_2 and instantiations i_1, i_2 such that $i_1 \in \text{match}(h_1, h_2)$, $i_2 \in \text{match}(t_1, t_2)$ and $\text{areMergeableInsts}(i_1, i_2, i)$.

From the induction hypothesis, the following can be inferred from the two recursive calls to *match*:

$$\text{instantiate}(h_1, i_1) =_{\alpha} h_2 \tag{6.1}$$

$$\text{instantiate}(t_1, i_2) =_{\alpha} t_2 \tag{6.2}$$

Applying Lemma 6.1 to (6.1) and Corollary 6.1 to (6.2), yields the following following equations:

$$\text{instantiate}(h_1, i) =_{\alpha} \text{instantiate}(h_1, i_1) =_{\alpha} h_2 \tag{6.3}$$

$$\text{instantiate}(t_1, i) =_{\alpha} \text{instantiate}(t_1, i_2) =_{\alpha} t_2 \tag{6.4}$$

Using the definition of *instantiate*, then applying the equations (6.3) and (6.4) to the result we can deduce that:

$$\begin{aligned} & \textit{instantiate}(\langle h_1 \rangle \hat{\wedge} t_1, i) \\ &= \langle \textit{instantiate}(h_1, i) \rangle \hat{\wedge} \textit{instantiate}(t_1, i) \\ &=_{\alpha} \langle h_2 \rangle \hat{\wedge} t_2 \end{aligned}$$

6.4.2.5 Non-parametric functions

Given a non-parametric function application of the form $p = \text{fnApplic}(f, as)$, for queries q and instantiations i such that $i \in \text{match}(p, q)$ then from Alg. 1 on page 126 it can be deduced that $q = \text{fnApplic}(f, as')$ and $i \in \text{match}(as, as')$. From the definition of *instantiate*, and the induction hypothesis it follows that:

$$\begin{aligned} & \textit{instantiate}(\text{fnapplic}(f, as), i) \\ &= \text{fnapplic}(f, \textit{instantiate}(as, i)) \quad [\text{Defn. of instantiate}] \\ &=_{\alpha} \text{fnapplic}(f, as') \quad [\text{Induction hypothesis}] \end{aligned}$$

6.4.2.6 Parametric functions

Given a parametric function of the form $p = \text{termparam}(f, es)$, then from the *match* algorithm $i \in \text{match}_{\text{funct}}(f, es, q)$. From the implementation of *match_{funct}* (see Alg. 5) it can in turn be deduced that there exists instantiations i_1 and i_2 , and a term e' such that $(e', i_1) \in \text{replacement}(q, es)$, $\text{freeVars}(e') = \emptyset$, $i_2.\text{finsts} = \{f \mapsto e'\}$ and $\text{areMergeableInsts}(i_1, i_2, i)$.

$$\begin{aligned} & \textit{instantiate}(p, i) \\ &= \textit{instantiate}(\text{termparam}(f, es), i) \\ &= \textit{substPlaceHolders}(e', \\ & \quad \{k : 1 \dots \#es \bullet k \mapsto \textit{instantiate}(es(k), i)\}) \quad [\text{Defn. of instantiate}] \\ &= \textit{instantiate}(\textit{substPlaceHolders}(e', es), i) \quad [\text{Lemma 6.6}] \\ &= \textit{instantiate}(\textit{substPlaceHolders}(e', es), i_1) \quad [\text{Corollary 6.2}] \\ &=_{\alpha} q \quad [\text{Lemma 6.5}] \end{aligned}$$

6.4.2.7 Quantified formulae

Given quantified formulae $p = \text{quant}(qu, vs_1, f_1)$ then from Alg. 6 $q = \text{quant}(qu, vs_2, f_2)$, $\#vs_1 = \#vs_2$ and for each $i \in \text{match}_{\text{quant}}(vs_1, f_1, vs_2, f_2)$ there are instantiations $i_1 \in \text{match}_{\text{mvs}}(vs_1, vs_2)$ and $i_2 \in \text{match}(f'_1, f'_2)$ where $f_1 =_{\alpha} f'_1$, $f_2 =_{\alpha} f'_2$ and $\text{areMergeableInsts}(i_1, i_2, i)$.

To show that $\text{instantiate}(p, i) =_{\alpha} q$ it is sufficient to show $\text{instantiate}(vs_1, i) =_{\alpha} vs_2$ and $\text{instantiate}(f_1, i) =_{\alpha} f_2$.

The first of these requirements is proven as follows:

$$\begin{aligned} & \text{instantiate}(vs_1, i) \\ &= \text{instantiate}(vs_1, i_1) && \text{[by Lemma 6.1]} \\ &=_{\alpha} vs_2 && \text{[by induction hypothesis]} \end{aligned}$$

The proof of the second requirement is as follows:

$$\begin{aligned} & \text{instantiate}(f_1, i) \\ &=_{\alpha} \text{instantiate}(f'_1, i) && \text{[by Lemma 6.7]} \\ &=_{\alpha} \text{instantiate}(f'_1, i_2) && \text{[by Cor. 6.1]} \\ &=_{\alpha} f'_2 && \text{[by induction hypothesis]} \\ &=_{\alpha} f_2 \square \end{aligned}$$

6.4.3 Completeness of algorithm

To prove completeness of the algorithms, it is necessary to show that every pattern p , query q and instantiation i satisfying $\text{matches}(p, q, i)$ has a “counterpart” returned by the match function. More precisely, for each p, q and i satisfying $\text{matches}(p, q, i)$, there is an instantiation $i_1 \in \text{match}(p, q)$ such that:

$$\text{instantiate}(p, i_1) =_{\alpha} \text{instantiate}(p, i).$$

Proving this requires some notion of a *minimal instantiation* i_1 of p , where intuitively i_1 can be generated from i by ensuring there are no free variables, and removing any redundant instantiations.

For example consider the pattern $p = f(x + y, g(z))$ and a query $q = x + y$. The instantiations p of which satisfy the matches definition include:

$$i = \{f(a, b) \rightsquigarrow a, g(a) \rightsquigarrow h\}$$

However h is an arbitrary term, and therefore there are infinite possible instantiations. In this case though, to describe the match between the pattern and query, the instantiation of g is not required (i.e. it is a redundant instantiation). An equivalent minimal instantiation in this case would be

$$i_1 = \{f(a, b) \rightsquigarrow a\}.$$

For the proof to proceed a formal definition of minimal instantiations would be required (outside of the scope of this thesis). Having provided a formal definition, it would then be a matter of showing that for a pattern p and a query q , there are finitely many minimal instantiations i_1 such that $instantiate(p, i_1) = q$ (this should follow from the formal definition), and the algorithm *match* generates all such i_1 's.

6.5 Associative commutative matching

Up to this point matching has been defined in terms of alpha-equivalence. In this section a relaxation of this is explored.

Consider the expressions $A \wedge B$ and $B \wedge A$ (where A and B are arbitrary logical formulae and \wedge is logical conjunction). These two expressions are logically equivalent, but in general will not be matched by the previous matching algorithm. That is matching up to alpha-equivalence is much stricter than matching up to logical equivalence.

It is desirable then to be able to relax the definition of matching, so that logically equivalent expressions are considered, rather than just alpha-equivalent expressions. However in general this is not possible without the aid of deductive reasoning support (in the form of some sort of theorem prover, or at least via term rewriting of some kind). However one class of expressions for which the basic algorithm can be extended without further deductive reasoning support is associative commutative (AC) expressions. This section describes how the basic algorithm can be extended to cover matching up to *AC-equivalence*.

6.5.1 AC-expressions

Associative and commutative operators are special classes of operators defined (over a set X) as follows:

Definition 6.1 *A function $\otimes : X \times X \rightarrow X$ is said to be associative if for all $u, v, w \in X$*

$$(u \otimes v) \otimes w =_{\alpha} u \otimes (v \otimes w).$$

A binary connective \otimes is said to be associative if for all formulae

$$(u \otimes v) \otimes w \Leftrightarrow u \otimes (v \otimes w).$$

Example 6.6 The operators $+$ and $*$ are both associative over the set of natural numbers \mathbb{N} . The binary connective \Rightarrow is associative.

Definition 6.2 *The operator $\otimes : X \times X \rightarrow X$ is said to be commutative if for all $u, v \in X$*

$$u \otimes v =_{\alpha} v \otimes u.$$

The relation $\otimes : X \leftrightarrow X$ is commutative if for all $u, v \in X$

$$u \otimes v \Leftrightarrow v \otimes u$$

Note that relations in CARE can be commutative, but not associative. The rest of Section 6.5 extends the matching algorithm for AC-functions. The algorithms could also easily be extended to cover AC binary connectives, and commutative relations.

6.5.2 Flattening expressions

Expressions can be *flattened*, with respect to a given AC operator, by modifying the abstract syntax to allow two or more branches in the syntax tree below the operator (so in effect “removing the bracketing”). For example the term $((a + b) + c)$ can be flattened to give $a + b + c$. The function *flatten* takes an AC-function and a term, and flattens the term with respect to the AC-function.

$\frac{\text{flatten} : \text{FunctionName} \times \text{Term} \rightarrow \text{seq Term}}{\text{flatten}(f, e) = \text{if } e = \text{fnApplic}(f, as) \text{ then } \text{flatten}(f, as) \text{ else } \langle e \rangle}$	
$\frac{\text{flatten} : \text{FunctionName} \times \text{seq Term} \rightarrow \text{seq Term}}{\text{flatten}(f, \langle \rangle) = \langle \rangle \quad \text{flatten}(f, \langle h \rangle \hat{\ } t) = \text{flatten}(f, h) \hat{\ } \text{flatten}(f, t)}$	

Note that if the outermost term is not the AC-operator in question, then no flattening will be done. For example the term $g((a + b) + c)$ will remain unchanged when flattened with respect to $+$.

6.5.3 AC-equivalence

Two terms t_1 and t_2 are said to be AC-equivalent with respect to a set of AC operators ns , denoted $AC_equiv(ns, t_1, t_2)$, if there is some permutation of the AC-subterms of t_1 and t_2 , such that the resulting terms are the same (up to renaming of bound variables):

$\frac{AC_equiv : \mathbb{P} OpName \rightarrow (\text{Term} \leftrightarrow \text{Term})}{\forall ns : \mathbb{P} OpName; t_1, t_2 : \text{Term} \bullet AC_equiv(ns, t_1, t_2) \Leftrightarrow (t_1 = \text{var } x \wedge t_2 = \text{var } x) \vee (t_1 = \text{ph } j \wedge t_2 = \text{ph } j) \vee (t_1 = \text{fnApplic}(f, as_1) \wedge t_2 = \text{fnApplic}(f, as_2) \wedge (f \in ns \wedge \exists as'_1, as'_2 : \text{seq Term} \bullet as'_1 = \text{flatten}(f, as_1) \wedge as'_2 = \text{flatten}(f, as_2) \wedge \#as'_1 = \#as'_2 \wedge \exists \pi : \text{dom } as'_1 \twoheadrightarrow \text{dom } as'_1 \bullet \forall (i, j) \in \pi \bullet AC_equiv(ns, as'_1(i), as'_2(j)))) \vee (f \notin ns \wedge \#as_1 = \#as_2 \wedge \forall i : 1 \dots \#as_1 \bullet AC_equiv(ns, as_1(i), as_2(i)))}$	
--	--

Example 6.7 Given the operator $+$ which is associative commutative over the set of natural numbers, then the following expressions are AC-equivalent:

$$(a + b) + c, a + (b + c), a + (c + b), b + (a + c), \dots \square$$

AC-equivalence is less strict than alpha-equivalence in that any terms that are alpha-equivalent will also be AC-equivalent; the reverse is not true. AC-equivalence is however stricter than logical equivalence, in particular any AC-equivalent expressions are logically equivalent.

6.5.4 Specification of AC-matching

A pattern p is said to match a query q up to AC-equivalence, with respect to a set of AC operators, if there exists an instantiation i , such that the result of instantiating p under i is AC-equivalent (with respect to the operator set) to q , i.e.:

$$\left| \begin{array}{l} \text{matches}_{AC} : (\mathbb{P} \text{ OpName} \times \text{Term} \times \text{Term} \times \text{Inst}) \\ \hline \forall ns : \mathbb{P} \text{ OpName}; p, q : \text{Term}; i : \text{Inst} \bullet \\ \text{matches}_{AC}(ns, p, q, i) \Leftrightarrow AC_equiv(ns, \text{instantiate}(p, i), q) \end{array} \right.$$

Example 6.8 Suppose that \otimes is an AC operator, and that g is a parametric function. Then the pattern $p == (g(x, y) \otimes z) \otimes h(c)$ is equivalent to the query $q == (x \otimes h(c)) \otimes (z \otimes y)$ under the instantiation $I == \{g(a, b) \rightsquigarrow a \otimes b\}$. To show this is true, firstly instantiate p with I , to give the term p^I :

$$\begin{aligned} p^I &= \text{instantiate}(p, I) \\ &= (\text{instantiate}(g(x, y), I) \otimes z) \otimes h(c) \\ &= ((x \otimes y) \otimes z) \otimes h(c) \end{aligned}$$

Next flatten q and p^I with respect to \otimes to get the lists $l_1 = \langle x, h(c), z, y \rangle$ and $l_2 = \langle x, y, z, h(c) \rangle$. Finally generate the permutation, π , of the elements of l_2 , such that $\forall i : 1..4 \bullet l_1(i) = l_2(\pi(i))$. This is satisfied by the following permutation:

$$\pi = \{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 3, 4 \mapsto 2\}.\square$$

6.5.5 An algorithm for AC-matching

The match algorithm can be extended by considering AC operators, and matching up to AC-equivalence. The $match_{AC}$ function extends $match$ for AC-functions. The input to $match_{AC}$ includes a set of AC operators as well as a pattern and a query.

$$\left| \text{match}_{AC} : \mathbb{F} \text{ OpName} \times \text{Term} \times \text{Term} \rightarrow \mathbb{F} \text{ Inst} \right.$$

The function $match_{AC}$ can be implemented in a similar manner to $match$ (see Section 6.3.3), except each call to $match_{AC}$ takes a set of operator names (the AC operators), and an extra case is considered. That is if the pattern p is of the form $p = f(a_1, \dots, a_m)$ and the query q is of the form $t = f(b_1, \dots, b_n)$, where f is a non-parametric, AC-function, then the algorithm for matching p against q proceeds as follows:

1. Apply the function $flatten$ to the argument lists a_1, \dots, a_m and b_1, \dots, b_n of the pattern and query respectively, to get the lists $l_p = a'_1, \dots, a'_j$ and $l_q = b'_1, \dots, b'_k$, where $j \geq m, k \geq n$;
2. Remove any duplicates from the lists l_p and l_q , i.e. any identical terms that appear in both lists, to give the lists rl_p and rl_q respectively;
3. Generate the set of partitions of the list rl_q , into a list of bags bl such that:

- (a) $\#bl = \#rl_p$

- (b) $\sum_{k=1}^{\#bl} count\ bl(k) = \#rl_q$

- (c) for each $k \in 1 \dots \#rl_p$

$$count\ bl(k) = \begin{cases} \geq 1 & \text{if } rl_p(k) \text{ is a parametric function application} \\ 1 & \text{otherwise} \end{cases}$$

- (d) for each $k \in 1 \dots \#rl_q$ there exists $j \in 1 \dots \#bl$ such that $rl_q(k)$ in $bl(j)$

- (e) each element appears the same number of times in the list of bags bl that it appeared in the list rl_q

4. For each partition list bl generated in the previous step, form the list of terms pl_q by mapping each bag to a term as follows:

- (a) for bags with a single element of the form $\llbracket u \rrbracket$ return the term u

- (b) for bags $\llbracket u_1, \dots, u_v \rrbracket$ with more than one element return the term

$$f(u_1, \dots, u_v)$$

5. Apply the function $match(rl_p, pl_q)$ to get a set of matches for each list of terms pl_q generated in step 4.

The match algorithm can be extended in a similar manner for non-parametric AC-relation applications.

Example 6.9 Given the pattern $p == (g(x, y) \otimes z) \otimes h(c)$ where \otimes is an AC-function and g is a parametric function, and query $q == (x \otimes h(c)) \otimes (z \otimes y)$ from Ex. 6.8. Then the algorithm for generating matches for p and q proceeds as follows.

1. flatten p and q with respect to the AC-operator \otimes to give the lists:

$$\begin{aligned} l_p &= \langle g(x, y), z, h(c) \rangle \\ l_q &= \langle x, h(c), z, y \rangle \end{aligned}$$

2. Remove elements which appear in both lists to give the lists:

$$\begin{aligned} rl_p &= \langle g(x, y) \rangle \\ rl_q &= \langle x, y \rangle \end{aligned}$$

3. Partition rl_q to give the list of bags $\langle [x, y] \rangle$
4. Form the list of terms $pl_q = \langle x \otimes y \rangle$
5. Generate the set of matches for $match(\langle g(x, y) \rangle, \langle x \otimes y \rangle)$. The result consists of the single instantiation $g(a, b) \rightsquigarrow a \otimes b$.

6.6 Type-constrained matching

The basic matching algorithm given earlier in the chapter is purely structure-based, in that terms which have the same structure but have different (mathematical) types will return a match. The aim of the following section is to describe a narrowing to the basic algorithm which will eliminate matches between terms with incompatible types.

This technique, referred to as *type-constrained matching* is developed in two stages. The first stage is where matches between expressions with incompatible

types are dismissed by attempting to unify the signatures of the pattern and query. In the second stage the algorithm is enhanced further by attempting to infer extra information about type parameters by comparing the expected function signatures with the actual signatures. Finally a specification for type-constrained matching is given which incorporates the two stages of the algorithm.

6.6.1 Preliminaries

Recall that for each function and parametric function, an associated signature is given describing the types of the domain and codomain, as well as any polymorphic parameters. Let's suppose the following "lookup" function returns the signature of a function (overloaded for parametric and non-parametric functions):

$$\left| \begin{array}{l} \textit{signature} : \textit{FunctionName} \rightarrow \textit{FunctionSig} \\ \textit{signature} : \textit{FunctionParam} \rightarrow \textit{FunctionSig} \end{array} \right.$$

The function *output_signature* returns the mathematical type of a given term. The function takes as input a term, and a mapping which associates a type with any variable in the term. For variables *output_signature* returns the type of the variable; for functions and parametric functions *output_signature* returns the type of the codomain of the function.

$$\left| \begin{array}{l} \textit{output_signature} : \textit{Term} \times (\textit{Var} \mapsto \textit{Set}) \times \textit{seq Set} \rightarrow \textit{Set} \\ \textit{output_signature}(\textit{var } v, \textit{vm}, \textit{phm}) = \textit{vm}(v) \\ \textit{output_signature}(\textit{ph}(n), \textit{vm}, \textit{phm}) = \textit{phm}(n) \\ \textit{output_signature}(\textit{fnapplic}(f, \textit{as}), \textit{vm}, \textit{phm}) = \textit{signature}(f).\textit{codomain} \\ \textit{output_signature}(\textit{termparam}(p, \textit{as}), \textit{vm}, \textit{phm}) = \textit{signature}(p).\textit{codomain} \end{array} \right.$$

For functions and parametric functions an *input_signature* can also be defined, which describes the sets of the arguments of the function. This is given by the domain of the function signature.

It is assumed that this signature information is available for any function. Either the function has been declared within the context of the user's program, or it is part of the predefined CARE Mathematical toolkit.

While queries can't contain parameters, the signatures of the functions which appear in queries may be parameterised. Since the signatures of patterns may also contain parameters, the signatures of patterns and queries must be *unified*, rather than just matched. For example, suppose the query $concat(x, y)$ has a signature $seq X \times seq X$, where X is a parameter, and a pattern $f(y, x)$ has the signature $Y \times Z$, where Y and Z are parameters, then these signature require unification.

A pattern p unifies with a query q if there is an instantiation i (also referred to as a *unifier* in this context), such that:

$$\left| \begin{array}{l} \text{unifies} : \mathbb{P}(Set \times Set \times Inst) \\ \hline \forall p, q : Set; i : Inst \bullet \text{unifies}(p, q, i) \Leftrightarrow \\ \text{instantiate}(p, i) =_{\alpha} \text{instantiate}(q, i) \end{array} \right.$$

A function for returning the set of unifiers between two sets is required:

$$\left| \text{unify} : Set \times Set \rightarrow \mathbb{F} Inst \right.$$

Note that the abstract syntax for *Sets* is “first order” in the sense that type parameters are simple metavariables, thus first-order unification is sufficient. More details on unification of terms and formulae (in particular higher-order unification) are given in Section 6.8. It is assumed that a similar function for lists of sets, function signatures and relation signatures is also available.

$$\left| \begin{array}{l} \text{unify} : seq Set \times seq Set \rightarrow \mathbb{F} Inst \\ \text{unify} : FunctionSig \times FunctionSig \rightarrow \mathbb{F} Inst \\ \text{unify} : RelationSig \times RelationSig \rightarrow \mathbb{F} Inst \end{array} \right.$$

6.6.2 Specification of type-constrained matching

The aim of type-constrained matching is to eliminate matches between expressions with incompatible signatures: that is matches between a pattern and query whose signatures can't be unified. A pattern is said to be a type-constrained match with a query if the signatures are unifiable, and the expressions match in the original sense. The definition of $matches_{TC}$ requires the signatures of any free variables occurring in either the pattern or query. For terms the $matches_{TC}$ is defined as follows.

$$\begin{array}{l}
\overline{matches_{TC} : \mathbb{P}(Term \times Term \times (Var \mapsto Set) \times Inst)} \\
\forall p, q : Term; vm : Var \mapsto Set; i : Inst \bullet \\
\quad matches_{TC}(p, q, vm, i) \Leftrightarrow \\
\quad \quad instantiate(p, i) =_{\alpha} q \wedge \\
\quad \quad \forall f : FunctionParam; e : Term \bullet f \mapsto e \in i.finsts \Rightarrow \\
\quad \quad \quad unifies(signature(f).codomain, out_signature(e, vm, \\
\quad \quad \quad \quad signature(f).domain), i)
\end{array}$$

The function $match_{TC}$ which takes a query and pattern, and returns the set of matches satisfying the relation $matches_{TC}$ could be implemented by either adding a post-matching check to the basic algorithm, or by incorporating types checks at each stage of the match algorithm. The advantage of the first approach would be that there is minimal re-engineering of the matching algorithm required, and it is easier to use an existing type-checking tool. The main advantage of the second approach is from the efficiency side, since matches which aren't type compatible are ruled out much earlier in the process.

Example 6.10 Given a pattern of the form $f(x, y)$, where f is a parametric function with signature $\mathbb{F} E \times E \rightarrow \mathbb{F} E$, and a query $append(x, y)$ where $append$ has signature $X \times seq X \rightarrow seq X$. Then the original $match$ function (as defined in Section 6.3), registers a match between the pattern and query with instantiation $f(a, b) \rightsquigarrow append(a, b)$. However the output-signatures of the pattern and query (i.e. $\mathbb{F} E$ and $seq X$ respectively) are not unifiable. \square

6.6.3 Type-constrained matching algorithm

In the previous section the signatures of expressions were used to eliminate matches between expressions with incompatible types. This approach can be extended by using the signatures to infer more information about parameters. That is, since any set parameters used in the signatures of other parameters are instantiated eventually, information about the signatures of query and pattern terms (and subterms) can be used to attempt to instantiate these set parameters.

This extra information can come from two sources. The first is from unifying the output-signatures of the query and pattern and retaining any instantiations

of parameters used in the pattern, rather than just determining whether the two signatures are unifiable as was done in the previous section.

The second source of this information comes from cross-checking the expected signature against the actual signature for any function or parametric function called in the query or pattern. The expected signature is just the input-signature for that particular function as was defined earlier. The actual signature is calculated by working out the output-signature for each of the arguments in the function call.

$$\mid \text{match}_{TC} : Term \times Term \times (Var \rightarrow Set) \rightarrow \mathbb{F} Inst$$

Example 6.11 Given a pattern $hd(base, x)$ and a query $append(x, nil)$ where x has type \mathbb{N} . Suppose that hd and $base$ are parameters with signatures:

$$\begin{aligned} hd &: A \times E \rightarrow A \\ base &: () \rightarrow A \end{aligned}$$

where A and E are parametric sets. Furthermore suppose that $append$ and nil have signatures:

$$\begin{aligned} append &: X \times \text{seq } X \rightarrow \text{seq } X \\ nil &: () \rightarrow \text{seq } Y \end{aligned}$$

where X and Y are parametric sets.

The basic matching algorithm returns a match between the pattern and query with instantiation:

$$hd(a, b) \rightsquigarrow append(b, a), base \rightsquigarrow nil.$$

Next the output-signatures of the pattern and query are unified. For this example the signature of hd is unified against the signature of $append$, and similarly the signature of $base$ against that of nil . That is unify A with $\text{seq } X$ and A with $\text{seq } Y$. This can be achieved with the instantiation $X \rightsquigarrow Z, A \rightsquigarrow \text{seq } Z, Y \rightsquigarrow Z$ where Z is a new parametric set.

Now attempt to unify the expected signature against the actual signature for the various function calls in the query and the pattern. Firstly the call to $append$ has an expected input signature of $X \times \text{seq } X$, while the actual input signature

(from the arguments of *append*) is $\mathbb{N} \times \text{seq } Y$. These signatures can be unified with instantiations $X \rightsquigarrow \mathbb{N}$, $Y \rightsquigarrow \mathbb{N}$.

Similarly for the call to *hd*, the expected signature is $A \times E$, the actual signature is $A \times \mathbb{N}$. These signatures can be unified with instantiation $E \rightsquigarrow \mathbb{N}$.

Merging the instantiations from the various steps results in the instantiation:

$$\{hd(a, b) \rightsquigarrow append(b, a), base \rightsquigarrow nil, A \rightsquigarrow \text{seq } \mathbb{N}, E \rightsquigarrow \mathbb{N}, X \rightsquigarrow \mathbb{N}, Y \rightsquigarrow \mathbb{N}\}$$

□

An implementation of $match_{TC}$ is given in Alg. 7. For variables and placeholders the algorithm is the same as that for the original match function on page 126. Non-parametric function applications are matched by matching the corresponding argument lists (as done previously), however the algorithm is extended by unifying the expected signatures of the functions against the actual signatures of the arguments. For parametric function patterns, the auxiliary function $match_{TC_{\text{funct}}}$ (implemented in Alg. 8 below) is called. The implementation of $match_{TC_{\text{funct}}}$ on page 129 is similar to that of $match_{\text{funct}}$ except that the signature of the pattern is matched against that of the query. The function $replacement_{TC}$ is the same as the $replacement$ function except it calls $match_{TC}$ recursively rather than $match$.

6.7 Interactive matching

Matching expressions can typically result in a number of possible matches. The approach discussed so far has been to generate all possible matches for a given query and pattern and then present the set of matches to the user. This approach however can sometimes be undesirable, particularly when there are a large number of matches involved.

An alternative solution is to allow the user to interact with the search process. After each possible match is found, the user is queried as to whether this is a suitable match, or whether they wish to continue the search process. This allows the user to jump out of search process earlier once a suitable match has been found. Other

Algorithm 7 Calculate $is = match_{TC}(p, q, vm)$ for terms

```

1:  $is := \emptyset$ 
2: case  $p$  of
3:   var  $x$ :
4:     if  $q = \text{var } x$  then
5:        $is := \{trivInst\}$ 
6:   ph  $j$ :
7:     if  $q = \text{ph } j$  then
8:        $is := \{trivInst\}$ 
9:   fn $Applic(f, as_1)$ :
10:    if  $q = \text{fn}Applic(f, as_2)$  then
11:       $is_1 := match_{TC}(as_1, as_2, vm)$ 
12:       $is_2 := unify(signature(f).domain,$ 
13:         $\{j : 1.. \#as_1 \bullet j \mapsto out\_signature(as_1(j), vm)\})$ 
14:       $is_3 := unify(signature(f).domain,$ 
15:         $\{j : 1.. \#as_2 \bullet j \mapsto out\_signature(as_2(j), vm)\})$ 
16:      for all  $i_1 \in is_1, i_2 \in is_2, i_3 \in is_3$  do
17:        if  $mergeableInsts(i_1, i_2, i_3) \wedge mergeableInsts(i_3, i_4, i)$  then
18:           $is := is \cup \{i\}$ 
19:    term $param(f, as)$ :
20:       $is := match_{TC}^{fn} (f, as, q, vm)$ 

```

Algorithm 8 Calculate $is = match_{TC}^{fn}(f, es, e, vm)$

```

1:  $is := \emptyset$ 
2:  $rs := replacement_{TC}(e, as, vm)$ 
3:  $is_1 := unify(signature(f).codomain, out\_signature(e, vm))$ 
4: for all  $(e', i_1) \in rs$  such that  $freeVars(e') = \emptyset$  do
5:   for all  $i_2 \in is_1$  do
6:      $i_3.finsts := \{f \mapsto e'\}, i_3.rinsts := \emptyset, i_3.sinsts := \emptyset$ 
7:     for all  $i \in mergeInstSet(\{i_1, i_2, i_3\})$  do
8:        $is := is \cup \{i\}$ 

```

solutions, including providing the user with guidance during the search process are discussed in Section 8.5.1.

6.7.1 A formal design

It is assumed that the function $match_*$ is defined. It is similar to the function $match$, except for a given pattern and query it returns a list of matches, rather than a set. Each element of this list satisfies the $matches$ relationship for a given pattern and query:

$$\mid match_* : Term \times Term \rightarrow seq Inst$$

Here it is assumed that the function $match_*$ uses *lazy evaluation* to generate the list of instantiations. That is each member of the list is only evaluated once it is actually required. It will be shown how this works in the context of interactive matching later in this section.

At each stage in the matching process the user is queried for a response. Three basic responses are modelled: accept the current match and continue; reject the current match and continue; or stop the match process with the current match:

$$Response ::= accept \mid reject \mid stop$$

Given a sequence of possible matches, together with a sequence of responses, (where the list of matches has at least as many members as the responses list), the function $generate_next_match$ is defined, which returns the set of user-accepted matches. This set is calculated by getting the next match from the list, together with the next response. If the response is “accept”, then the matching process is continued and the current match is added to the set of results. If the response is “reject”, then the process is continued with no addition to the results set. Otherwise the process is terminated with the current match returned.

Note that like $match_*$, the responses are evaluated lazily, only being generated after each instantiation is generated.

$$\begin{array}{l}
\hline
generate_next_match : seq Inst \times seq Response \rightarrow \mathbb{F} Inst \\
\hline
generate_next_match(\langle \rangle, \langle \rangle) = \emptyset \\
\forall i : Inst; is : seq Inst; r : Response; rs : seq Response \bullet \\
\quad generate_next_match(\langle i \rangle \hat{\ } is, \langle r \rangle \hat{\ } rs) = \\
\quad \text{if } r = \text{accept} \\
\quad \text{then } \{i\} \cup generate_next_match(is, rs) \\
\quad \text{else if } r = \text{reject} \\
\quad \text{then } generate_next_match(is, rs) \\
\quad \text{else } \{i\}
\end{array}$$

The function $match_{interact}$ takes a query, a pattern and a sequence of responses and returns a set of instantiations, corresponding to a subset of those formed by the ordinary match, by interacting with the user and giving them the option to quit after each match is found. It makes use of the $generate_next_match$ and $match_*$ functions:

$$\begin{array}{l}
\hline
match_{interact} : Term \times Term \times seq Response \rightarrow \mathbb{F} Inst \\
\hline
\forall q, t : Term; rs : seq Response \bullet \\
\quad match_{interact}(q, t, rs) = generate_next_match(match_*(q, t), rs)
\end{array}$$

6.8 Unification

With the exception of unification of signatures in Section 6.6, it has been assumed that only the pattern contains parameters, and thus pattern matching or one-way unification has been sufficient. While this is generally much simpler than unification, it is often useful to be able to parameterise the query as well as the pattern. Unification will be required in later parts of this thesis.

A pattern p unifies with a query q if there is an instantiation i (also referred to as a *unifier* in this context), such that p instantiated under i is alpha-equivalent to q .

$$\begin{array}{l}
\hline
unifies : \mathbb{P}(Term \times Term \times Inst) \\
\hline
\forall (p, q, i) \in unify \bullet instantiate(p, i) =_{\alpha} instantiate(q, i)
\end{array}$$

Many algorithms have been implemented since Huet [44] first described an algorithm for higher-order unification. It is not the purpose of this thesis to add any

innovations to the problem of higher-order unification, but rather just to describe how unification can be used in the context of developing retrieval mechanisms for formally specified components.

The Prolog language contains first-order (untyped) meta-variables. Deduction in Prolog is based on so-called standard (first-order) unification of these meta-variables. Instances of terms are obtained by substituting one or more of the variables in the term with new subterms, e.g. $f(a, b)$ is an instance of $f(X, b)$ where the subterm a is substituted for the meta-variable X . Two terms are said to unify if they have a common instance, where variables common to both terms are substituted consistently.

In the case of higher-order logics (i.e. logics whose parameters range over functions or relations), standard unification is no longer applicable. Huet [44], was first to propose an algorithm for unifying higher-order terms in typed lambda-calculus. Since then many implementations of Huet's algorithm have been developed. It has been established that in general higher-order unification is undecidable (that is given arbitrary terms it is not always possible to determine when they are unifiable). However the unification of certain classes of higher-order terms have been found to be decidable. In particular for a class of terms discovered by Miller referred to as "patterns", unification has been shown to be decidable [67, 71]. Similarly third-order matching has been shown to be decidable [20], and there are certain classes of higher-order terms for which matching is decidable [94].

Higher-order unification has been most commonly applied to automated reasoning. The HOL theorem prover is "hardwired" to higher order logic, and provides built-in support for higher-order unification. On the other hand, the Isabelle theorem prover [72] is a generic theorem prover, designed for reasoning in a variety of formal theories. It does however provide support for higher-order logic, with a built-in algorithm for higher-order unification. Note that its (lazy) algorithm is not guaranteed to terminate, since the set of "most general unifiers" for terms in its metalanguage is not always finite.

Higher-order unification has also been utilised in programming languages. As

an example lambda-Prolog [67] extends standard Prolog, by allowing parameters to range over functions and relations. As a result reasoning in lambda-Prolog relies on higher-order unification rather than standard unification.

Finally, another area of interest worth noting in the field of higher-order unification are the extensions made to Huet's original algorithm. Of particular interest to us is AC-matching and AC-unification, where the algorithm is extended to be able to unify AC-equivalent terms. The algorithm for AC-matching given in this chapter was in fact based on an algorithm by Lincoln and Christian [57].

For the purpose of exploring how unification can be applied to our problem, a very simple algorithm has been prototyped. This algorithm is based on the basic matching algorithm.

$$\mid \text{ unify} : \text{Term} \times \text{Term} \rightarrow \mathbb{F} \text{Inst}$$

For terms, unification of variables and non-parametric functions is the same as for matching. Similarly unifying a parametric function application with a non-parametric term is the same as matching. To unify two parametric function applications, the method used is to freeze one of the parameters, treating the term like a non-parametric function application and apply the matching algorithm. To get the remaining unifiers, the other parameter is in turn frozen.

The unification algorithm described above is by no means complete, but it is sufficient for investigating some of the issues later in the thesis. The soundness of the algorithm follows from the soundness of the underlying basic matching algorithm.

Example 6.12 To unify the terms $f(g(x, y))$ and $h(k(y, x))$, where f , g , h and k are parameters, one of the two terms is frozen. Firstly freeze the parameter h in the second term, instantiating $f(a) \rightsquigarrow h(a)$ and reducing the problem to unifying the $g(x, y)$ with $k(y, x)$.

To unify $g(x, y)$ and $k(y, x)$, again freeze one of the terms. Freezing the second term leads to the instantiation $g(a, b) \rightsquigarrow k(b, a)$. Similarly freezing the first term leads to the instantiation $k(a, b) \rightsquigarrow g(b, a)$. So by freezing the second term at the

top level there are two possible unifiers:

$$\{f(a) \rightsquigarrow h(a), g(a, b) \rightsquigarrow k(b, a)\}$$

$$\{f(a) \rightsquigarrow h(a), k(a, b) \rightsquigarrow g(b, a)\}$$

Similarly by freezing the parameter f at the top level the following unifiers result:

$$\{h(a) \rightsquigarrow f(a), g(a, b) \rightsquigarrow k(b, a)\}$$

$$\{h(a) \rightsquigarrow f(a), k(a, b) \rightsquigarrow g(b, a)\}$$

□

6.9 Discussion

In this chapter a number of techniques for matching mathematical expressions in CARE have been developed. Included were algorithms for matching up to alpha-equivalence and AC-equivalence. Also described were two techniques for narrowing the search space - i.e. type-constrained matching and interactive matching. Finally methods for unifying two parameterised expressions were discussed. These algorithms have all been implemented as part of a retrieval tool which has been integrated with the CARE tools.

The first algorithm, for matching expressions up to alpha-equivalence, is an adaptation of the algorithm used for pattern matching expressions in Mural [47].

The implementation of matching up to AC-equivalence was inspired by an algorithm proposed by Lincoln and Christie [57]. Type-constrained matching is based on *type-checking*, a technique commonly used to check type consistencies for formal languages (see for example the *fuzzZ* type-checker [86]). The idea of the user interacting with a search is similar to the rationale behind interactive theorem provers. The unification algorithm was not developed with efficiency or completeness in mind; instead merely as a means of allowing parameterised queries to be matched with a minimum of development effort. Many unification algorithms have been developed (as described in Section 6.8). The user is referred to the seminal paper by Huet [44] or other literature on unification for more details.

Implementing matching for both alpha-equivalence and AC-equivalence has enabled a comparison of the complexity and efficiency of algorithms for the two. AC-equivalence makes better use of the semantics of the language, and thus enables more matches to be found. However AC-equivalence requires a more complex algorithm, which in general will be less efficient than the simpler algorithm for alpha-equivalence. In general terms, there will be a tradeoff between the range of matches an algorithm will return and the complexity and efficiency of the algorithm. In particular the less strict an equivalence, the more complex the algorithm required. From the user's point of view, the main tradeoff is between efficiency and the "intelligence" of the algorithm. Rather than forcing this decision on the user one way or the other, a variety of algorithms could be integrated with a retrieval tool, with the user selecting one particular algorithm before conducting a search.

When integrated with a retrieval tool, type-constrained matching and interactive matching provide a means of narrowing the search space. As a result the retrieval tool can perform searches more efficiently, and return a smaller and more useful set of matches. The idea of interacting with the user during a search is very general and could be applied to most kinds of components. While the exact details of type-constrained matching will depend on the particular formal language, the fact that many formal languages have a type system, means that this idea is still widely applicable.