

Chapter 8

Matching Modules

8.1 Introduction

Given a library of reusable modules, it is important to have a retrieval tool which helps the user find a suitable module; particularly when the library is large and/or the user is unfamiliar with the structure and contents of the library. The aim of this chapter is to develop useful retrieval techniques, which fit well with the module adaptation techniques discussed in Chapters 3 and 4. However the results are quite general, and are largely independent of the nature of the individual units, and the techniques used to match these units.

To search for a suitable module (or template in the context of CARE), the specifications of one or more desired units are given (referred to as the *search query*). The search proceeds by matching the individual search query units against the template units for each template in the library. Matches are expressed in terms of adaptations to the module components.

Module matching relies on an algorithm for matching two sets of units: the first set, the search query, is composed of a collection of unit queries; the second set consists of the units contained in the module. Several strategies for matching sets of units will be explored, giving the user a choice between flexibility and precision of matching.

Another technique that is useful when conducting searches on a library of mod-

ules is a flexible level of interaction with the user. Interaction with the user during the search process allows the user to terminate the search once a suitable module has been found: the idea is similar to interactive matching at the expression level (see Section 6.7). No interaction during the search process means that the search is run until completion, upon which time the results are summarised to the user. Both of these situations are useful depending on the requirements of the user. For this reason a solution is proposed in which the level of interaction can be set by the user, varying from no interaction up to the highest level of interaction, with several intermediate levels.

This chapter is structured as follows: Section 8.2 contains a description of the requirements for the retrieval tool, defining three different matching strategies: ALL-match, SOME-match and ONE-match. Given a search query consisting of a set of units, the three strategies can be used to find modules with units matching: each of the query units; at least one of the query units; and exactly one of the query units respectively.

Section 8.3 defines algorithms for each of these search strategies. Before describing these implementations a simpler class of problems are tackled. More precisely three strategies: AND-match, OR-match and XOR-match are implemented. These strategies can be used to match a query consisting of exactly two units. The AND strategy is used to search for modules matching both of the query units - it can be thought of as the two query counterpart of ALL-match. The XOR strategy is used to search for modules matching exactly one query unit - its general counterpart is the ONE-match strategy. The OR strategy is used to search for modules matching either one or both of the query units - its general counterpart is the SOME-match strategy. Note that there is no attempt to optimise the performance of these algorithms within this thesis.

Section 8.4 briefly describes how the general matching approach can be applied to matching of templates in CARE. The discussion includes a simple example.

Section 8.5 describes how the module matching algorithms can be used to build a generic search engine. The search engine is engineered in such a way that is quite

general, and as a result can be used as the basis for building a variety of application specific retrieval tools. (One particular application of this search engine, as described in Chapter 9, is building a retrieval tool for CARE).

8.2 Requirements analysis

This section discusses requirements for retrieval strategies for modules, and presents a framework for formally specifying such strategies.

8.2.1 Prerequisites

Types for representing units, adaptations of units and unit queries are assumed to have been defined.

$$[Unit, UnitAdapt, UnitQuery]$$

The notion of a unit matching a unit query under some adaptation is assumed to be defined. Note that while the unit matching approach for CARE units described in the previous chapter could be used here, other unit matching approaches could be substituted without loss of generality.

$$| \text{matches}_{unit} : \mathbb{P}(Unit \times UnitQuery \times UnitAdapt)$$

While the algorithms for matching modules given in this section are independent of the kind of underlying units, a function for matching individual units, shall be assumed.

$$\left| \begin{array}{l} \text{match}_{unit} : Unit \times UnitQuery \rightarrow \mathbb{F} UnitAdapt \\ \hline \forall u : Unit; q : UnitQuery; i : UnitAdapt \bullet \\ i \in \text{match}_{unit}(u, q) \Rightarrow \text{matches}_{unit}(u, q, i) \end{array} \right.$$

A function for merging adaptations is also required.

$$| \text{mergeUnitAdapt} : UnitAdapt \times UnitAdapt \rightarrow UnitAdapt$$

Assumption 8.1 *It is assumed that unit adaptation is monotonic with respect to merging: i.e.*

$$(i_1, i_2) \in \text{dom } \text{mergeUnitAdapt} \wedge \\ (\text{matches}_{\text{unit}}(u, q, i_1) \vee \text{matches}_{\text{unit}}(u, q, i_2)) \\ \Rightarrow \text{matches}_{\text{unit}}(u, q, \text{mergeUnitAdapt}(i_1, i_2)).$$

It is assumed that modules have been formally defined; however no assumptions are made about the structure of modules.

[*Module*]

The function *unitsOf*, which returns the set of units contained within a module is also required.

$$| \text{unitsOf} : \text{Module} \rightarrow \mathbb{F} \text{Unit}$$

8.2.2 Adapting Modules

It is assumed here that modules can be adapted by adapting individual units, and by subsetting (see Chapter 4). Recall that a single unit adaptation is used to represent adaptation of the unit set. This ensures that adaptations such as parameter instantiation and identifier renaming are applied consistently throughout the entire module (see Section 4.4 for more details).

$$\text{ModAdapt} == \text{UnitAdapt} \times \mathbb{F} \text{Unit}$$

8.2.3 Module Retrieval

The library is searched by giving a module query which consists of a set of unit queries.

$$\text{ModQuery} == \mathbb{F} \text{UnitQuery}$$

The retrieval tool is based on *matching* a set of unit queries against a module (so-called *module matching*). In the remainder of this section three different strategies for module matching: ALL-match, SOME-match and ONE-match are described. Each of these strategies matches a subset of the template units - however the ALL-match strategy will generally return the larger subset.

8.2.3.1 ALL-match

The first module matching strategy given here is ALL-match, in which each of the unit queries must be matched against a module unit.

$$\left| \begin{array}{l} \text{matches}_{all} : \mathbb{P}(\text{Module} \times \text{ModQuery} \times \text{ModAdapt}) \\ \hline \forall m : \text{Module}; qs : \text{ModQuery}; a : \text{ModAdapt} \bullet \\ \text{matches}_{all}(m, qs, a) \Leftrightarrow \\ \quad \forall q \in qs \bullet \exists u : \text{Unit} \bullet u \in \text{unitsOf}(m) \wedge \\ \quad \text{matches}_{unit}(u, q, \pi_1 a) \wedge u \in \pi_2 a \end{array} \right.$$

Note that the *ModAdapt* argument is underconstrained. In particular the part of the adaptation describing which module units to include may contain extra units beside those matched. This reflects the fact that including subsets of a module may require extra units from the module to be included to ensure a self-contained subset. Therefore a module adaptation containing just those units in the module that were matched could be extended by including extra units to give a self-contained subset, giving an equivalent adaptation.

This search strategy is useful when the query units are closely related, and the user wants to maintain some sort of consistency between the units. For example the user may desire a number of components for manipulating a given data structure in a number of ways. To ensure that all of the components are consistently implemented (i.e. built on top of the same primitive data structure etc.), the user can include specifications for each component in the search query and do an ALL search. This will ensure that only templates that implement each of these components will be returned.

Suppose the user's program consists of a type representing the natural numbers, and fragments for adding and multiplying two naturals (all specified-only). To ensure these units are implemented by the same underlying data structure, they could be grouped together in a search query (see Fig. 8.1). A search can then be done using the ALL-match strategy to find templates which match each of the query units. One such candidate is the `Natural numbers` template in Fig. C.5.

This strategy is the most precise way of matching modules and will typically return the least number of matches. By increasing the number of unit queries, the

Type `Nat` has specification: \mathbb{N}

Fragment `n:Nat + m:Nat` has specification:
output `r:Nat` such that $r = m + n$

Fragment `n:Nat * m:Nat` has specification:
output `r:Nat` such that $r = m * n$

Figure 8.1: A search query for finding natural number primitives

precision is increased and as a result the number of matches is decreased. While the precision of this kind of search is relatively high, the flexibility is lower; the user will miss out on any modules that match most but not all of the unit queries.

8.2.3.2 SOME-match

Matching *some* unit queries against a module is a relaxation of the much stricter ALL-match strategy given in the previous section. Of interest here are modules which contain units which match a nonempty subset of the query set. This will include the set of matches formed by the ALL-match strategy. As would be expected, this kind of search is less precise than the previous one, and results in more matches being found. In contrast to ALL-match where an increase in the number of unit queries results in a decrease in matches; for the SOME-match an increase in the number of unit queries results in an increase in the number of matches. For this reason the user needs to be somewhat judicious in the number of units in the query given for this kind of match.

$$\begin{array}{|l}
 \hline
 matches_{some} : \mathbb{P}(Module \times ModQuery \times ModAdapt) \\
 \hline
 \forall m : Module; qs : ModQuery; a : ModAdapt \bullet \\
 matches_{some}(m, qs, a) \Leftrightarrow \\
 \exists ss \subseteq qs \bullet ss \neq \emptyset \wedge matches_{all}(m, ss, a)
 \end{array}$$

Consider development of programs using a step-wise refinement approach as used in systems such as CARE and B. The starting point is a specification of the desired component, represented by one or more unit specifications. At each stage in the development one or more of the specified-only units are implemented by other (more concrete) units, with these new units added to the program. The process continues

until all units in the program are implemented. To semi-automate each stage in the development a search of a library of predefined modules could be done to find units satisfying the specifications of the unimplemented units in the program. The search query is the set of specified-only units in the program. Attempting to match each of the query units (using the ALL-match strategy) is too strict a requirement in this case. Matching just some of the query units (i.e. using the SOME-match strategy) is sufficient, since the step-wise refinement approach only requires that at least one unit is implemented at each stage.

A similar situation occurs with theorem provers, where a set of conditions need to be discharged. At each stage in the proof one or more of these conditions must be proved, with each proof step possibly introducing new conditions. To automate the proof process a search tool could be used to find assertions from a library of theories which satisfy one or more of the conditions. The SOME-match strategy again could be used to find theories which satisfy some of these conditions.

8.2.3.3 ONE-match

A third strategy for module matching is to match exactly *one* query unit; this strategy is referred to as ONE-match. This can be thought of as being at the other end of the spectrum to matching all units. Note that the subset description part of the module adaptation contains exactly one unit.

$$\left| \begin{array}{l} matches_{one} : \mathbb{P}(Module \times ModQuery \times ModAdapt) \\ \hline \forall m : Module; qs : ModQuery; a : ModAdapt \bullet \\ matches_{one}(m, qs, a) \Leftrightarrow \\ \exists q \in qs; u : Unit \bullet u \in unitsOf(m) \wedge \\ matches_{unit}(u, q, \pi_1 a) \wedge \pi_2 a = \{u\} \end{array} \right.$$

The ONE-match strategy enables the user to include a number of alternate unit queries in order to find a single desired unit. This could be used when there are a number of equivalent ways of specifying a unit; for example the user might desire a unit for manipulating a list s , with a precondition stating that the list is nonempty - such a precondition could be given as either $\#s \neq 0$ or $s \neq \langle \rangle$. In this case two

queries could be formulated with the preconditions given above, but only one would be expected to match at a time.

Another example where this strategy is useful is in CARE where the specifications of fragments can contain a number of branches, each returning a different type of result. A query for finding such a branching fragment might consist of a number of unit queries representing each of the different ways of ordering the branches. Such equivalent specifications could be generated by a tool after the user has given a single specification. For example consider the two alternate specification for integer division given in Fig. 8.2, which treat the undefined case separate to other cases in the specification. To search for an implementation for integer division the user could give a query consisting of the two alternative specifications and use the ONE-match strategy to find a module matching exactly one of these unit queries.

Branching fragment `div1(a,b:Nat)` has specification:
if $b \neq 0$ then report `valid` with output `c:Nat`
such that $c = a/b$
else report `undefined`.

Branching fragment `div2(a,b:Nat)` has specification:
if $b = 0$ then report `undefined`
else report `valid` with output `c:Nat`
such that $c = a/b$.

Figure 8.2: Equivalent ways of specifying integer division

Matching one query component is like matching some, in that increasing the number of alternatives in the query decreases the precision and consequently increases the number of matches.

8.3 An algorithm for matching modules

8.3.1 Overview

The following section contains a discussion on how to implement the module matching routines ALL-match, SOME-match and ONE-match.

Before investigating these general algorithms a simpler set of problems are discussed: namely, matching query sets containing two units against modules. This can be done in three ways (strictly speaking there is a fourth way if the trivial case of matching neither of the unit queries is included). The first method, referred to as AND-match, succeeds if *both* unit queries match distinct module units. The second method, OR-match, succeeds when *at least one* of the unit queries matches a module unit. The third method, XOR-match, succeeds if *exactly one* of the unit queries matches a module unit.

Algorithms, described semi-formally, are given for each of these problems, together with a sketch of the proof showing that the algorithm meets its specification. A brief discussion on how the other methods can be implemented is also given.

Having discussed the simpler two-unit query problem, it is possible to generalise to an arbitrary unit query set. Indeed ALL-match can be seen as a generalisation of AND-match; both cases match all of the units in the query against distinct module units. As a result it is possible to extend the algorithm for AND-match to give an algorithm for ALL-match by using a similar strategy. Similarly SOME-match and ONE-match are generalisations of OR-match and XOR-match respectively, and the algorithms used for the simpler cases can be extended to the more general cases.

8.3.2 Matching Two-unit Queries Against Modules

In the following section a number of ways of matching a query consisting of two distinct units against a module are discussed.

8.3.2.1 AND-match

Given two query units q_1 and q_2 , then q_1 and q_2 match a module m , if there exist distinct units u_1 and u_2 , both members of m , such that q_1 matches u_1 and q_2 matches u_2 and the resulting adaptations are compatible.

$$\begin{array}{|l}
\hline
\text{matches}_{and} : \mathbb{P}(\text{Module} \times \text{UnitQuery} \times \text{UnitQuery} \times \text{ModAdapt}) \\
\hline
\forall m : \text{Module}; q_1, q_2 : \text{UnitQuery}; a : \text{ModAdapt} \bullet \\
\text{matches}_{and}(m, q_1, q_2, a) \Leftrightarrow \\
\exists u_1, u_2 \in \text{unitsOf}(m) \bullet u_1 \neq u_2 \wedge \{u_1, u_2\} \subseteq \pi_2 a \wedge \\
\text{matches}_{unit}(u_1, q_1, \pi_1 a) \wedge \text{matches}_{unit}(u_2, q_2, \pi_1 a) \\
\hline
\end{array}$$

The function $match_{and}$ returns the set of matches between two unit queries and a module, for which both queries are matched against distinct module units.

$$| \quad match_{and} : \text{Module} \times \text{UnitQuery} \times \text{UnitQuery} \rightarrow \mathbb{F} \text{ModAdapt}$$

Algorithm 12 gives an algorithm for implementing $match_{and}(q_1, q_2, m)$. For each distinct pair (u_1, u_2) of units of m , the unit-matching function is applied to yield two sets of possible instantiations, and the instantiations are merged pairwise from the two sets to yield adaptations.

Algorithm 12 Calculate $as = match_{and}(m, q_1, q_2)$

```

1:  $us := \text{unitsOf}(m)$ 
2:  $as := \emptyset$ 
3: for all  $u_1 \in us$  do
4:    $is_1 := \text{match}_{unit}(u_1, q_1)$ 
5:   for all  $u_2 \in us \setminus \{u_1\}$  do
6:      $is_2 := \text{match}_{unit}(u_2, q_2)$ 
7:     for  $i_1 \in is_1, i_2 \in is_2$  do
8:       if  $(i_1, i_2) \in \text{dom } \text{mergeUnitAdapt}$  then
9:          $as := as \cup \{(\text{mergeUnitAdapt}(i_1, i_2), \{u_1, u_2\})\}$ 

```

Proposition 8.1 *Each adaptation returned by the function $match_{and}$ agrees with the relation $matches_{and}$; i.e.*

$$\begin{array}{l}
\forall m : \text{Module}; q_1, q_2 : \text{UnitQuery}; a : \text{ModAdapt} \bullet \\
a \in match_{and}(m, q_1, q_2) \Rightarrow matches_{and}(m, q_1, q_2, a)
\end{array}$$

Following is a sketch of the proof for the above proposition.

Proof: Suppose that a is an adaptation returned by $match_{and}(q_1, q_2, m)$. From lines 4,6,7,9 of the algorithm, there exists $u_1, u_2 \in unitsOf(m)$ and instantiations i_1, i_2 such that $\pi_2 a = \{u_1, u_2\}$ and $i_j \in match_{unit}(u_j, q_j)$, $j = 1, 2$. From the specification of $match_{unit}$, it follows that:

$$matches_{unit}(u_1, q_1, i_1) \wedge matches_{unit}(u_2, q_2, i_2).$$

Since $\pi_1 a = mergeUnitAdapt(i_1, i_2)$, it follows by the monotonicity of instantiation (see Assumption 8.1) that $matches_{unit}(u_j, q_j, \pi_1 a)$ for $j = 1, 2$. Finally it follows from line 5 that $u_1 \neq u_2$, and hence that $matches_{and}(q_1, q_2, m, a)$. \square

8.3.2.2 XOR-match

The second kind of strategy for matching two-unit queries against modules is XOR-match. In this case we are interested in cases where the adaptation contains a single unit, and the unit matches one of the queries.

$$\left| \begin{array}{l} matches_{xor} : \mathbb{F}(Module \times UnitQuery \times UnitQuery \times ModAdapt) \\ \hline \forall m : Module; q_1, q_2 : UnitQuery; a : ModAdapt \bullet \\ matches_{xor}(m, q_1, q_2, a) \Leftrightarrow \\ \quad \exists u \in unitsOf(m) \bullet \\ \quad (matches_{unit}(u, q_1, \pi_1 a) \vee matches_{unit}(u, q_2, \pi_1 a)) \\ \quad \wedge \pi_2 a = \{u\} \end{array} \right.$$

The function $match_{xor}$ returns the set of matches between a two unit query and a module in which exactly one of the unit queries is matched.

$$\left| \begin{array}{l} match_{xor} : Module \times UnitQuery \times UnitQuery \rightarrow \mathbb{F} ModAdapt \end{array} \right.$$

An implementation for $match_{xor}$ is given in Algorithm 13.

Proposition 8.2 *Each adaptation returned by the function $match_{xor}$ agrees with the relation $matches_{xor}$; i.e.*

$$\begin{array}{l} \forall m : Module; q_1, q_2 : UnitQuery; a : ModAdapt \bullet \\ a \in match_{xor}(m, q_1, q_2) \Rightarrow matches_{xor}(m, q_1, q_2, a) \end{array}$$

The proof of this proposition is straightforward.

Algorithm 13 Calculate $as = match_{xor}(m, q_1, q_2)$

```

1:  $us := unitsOf(m)$ 
2:  $as := \emptyset$ 
3: for  $u \in us$  do
4:    $is_1 := match_{unit}(u, q_1)$ 
5:    $is_2 := match_{unit}(u, q_2)$ 
6:   for  $i \in is_1 \cup is_2$  do
7:      $as := as \cup \{(i, \{u\})\}$ 

```

8.3.2.3 OR-match

The third kind of matching strategy for two-unit queries is OR-match. In this case we are interested in cases where *one or both* of the query units match against distinct module units.

$$\begin{array}{|l}
 matches_{or} : \mathbb{P}(Module \times UnitQuery \times UnitQuery \times ModAdapt) \\
 \hline
 \forall m : Module; q_1, q_2 : UnitQuery; a : ModAdapt \bullet \\
 matches_{or}(m, q_1, q_2, a) \Leftrightarrow \\
 matches_{xor}(m, q_1, q_2, a) \vee matches_{and}(q_1, q_2, m, a)
 \end{array}$$

The function $match_{or}$ returns the set of matches between a two unit query and a module in which one or both of the unit queries are matched.

$$| \quad match_{or} : Module \times UnitQuery \times UnitQuery \rightarrow \mathbb{F} ModAdapt$$

The implementation of $match_{or}$ is given in Alg. 14. Alternatively $match_{or}$ could be “implemented” simply by combining the results of AND and XOR matching. However implementing OR-match separately can lead to some optimisations. Note that it is similar to that for $match_{and}$, except firstly at each stage the adaptations representing matches with single query units are also added to the accumulator. It is straightforward to show that the implementation of OR-match satisfies the specification.

Algorithm 14 Calculate $as = match_{or}(m, q_1, q_2)$

```

1:  $us := unitsOf(m)$ 
2:  $as := \emptyset$ 
3: for all  $u_1 \in us$  do
4:    $is_1 := match_{unit}(u_1, q_1)$ 
5:   for all  $u_2 \in us \setminus \{u_1\}$  do
6:      $is_2 := match_{unit}(u_2, q_2)$ 
7:     for  $i_1 \in is_1$  do
8:        $as := as \cup \{(i_1, u_1)\}$ 
9:     for  $i_2 \in is_2$  do
10:       $as := as \cup \{(i_2, u_2)\}$ 
11:     for  $i_1 \in is_1, i_2 \in is_2$  do
12:       if  $(i_1, i_2) \in \text{dom } mergeUnitAdapt$  then
13:          $as := as \cup \{(mergeUnitAdapt(i_1, i_2), \{u_1, u_2\})\}$ 

```

8.3.2.4 Completeness

Earlier it was noted that the *ModAdapt* argument in the *matches* relation has been left underconstrained. For example, given a module m and queries q_1 and q_2 , $matches_{and}$ considers not only those adaptations in which exactly two units are given in the adaptation, but also those in which other units have been added.

To reason about the completeness of the algorithms, some notion of an equivalence class of module adaptations must be considered. Two module adaptations are equivalent for a given module m , if

$$adapt_module(m, a_1) = adapt_module(m, a_2).$$

Intuitively two adaptations would be equivalent if they rename identifiers and variables, and instantiate parameters consistently and generated the same self-contained subset.

By partitioning the module adaptations into equivalence classes, completeness of the algorithms could be checked by proving that the algorithms return one adaptation from each applicable equivalence class.

8.3.3 Generalising the Approach

In this section the ideas of the previous section are extended to an arbitrary query set. Firstly the methods ALL-match, SOME-match and ONE-match, are generalised in terms of the two-unit query matching strategies AND-match, OR-match and XOR-match respectively. Then the HYBRID-match strategy, which is a hybridisation of the other strategies, is developed.

8.3.3.1 Implementing ALL-match

Matching all query components is a generalisation of the AND-match method given in the previous section. Here every query component must match a unique pattern component such that the corresponding adaptations are all mergeable.

$$| \quad match_{all} : Module \times ModQuery \rightarrow \mathbb{F} ModAdapt$$

$match_{all}$ can be implemented using a similar strategy to that used in the implementation given for $match_{and}$. The algorithm begins by attempting to match the first query unit against a unit from the module. If successful the second query unit is matched against a unit from the remainder of the module's unit set. This process continues until either all query unit have been matched or the algorithm fails to find a match at one of the stages. The complete set of matches can be generated by considering all combinations of query components against pattern components.

Recall that all of the adaptations of the individual matches must be mergeable to form a match between the query and module. Rather than doing all of these checks at the end of the process, the prototype algorithm accumulates an adaptation representing the merging of the adaptations generated so far. At each step in the process a check can be done to determine whether the adaptation representing the match between the current query and module is mergeable with the accumulated adaptation.

Another optimisation that can be made to develop an efficient algorithm, is to ensure that the set of matches between any pair of query and module units is generated *at most once*. One way of doing this is to firstly calculate matches between

any unit pairs, and store this information in a matrix for use in the main algorithm. This method ensures that each pair of units is matched *exactly once*.

However two units may never need to be matched, either because the algorithm never needs to check this, or in the case of interactive use the user may terminate the process before all pairs have been considered. The prototype solves this by checking for matches only when this information is first required; then storing the information for later reference.

8.3.3.2 SOME-match

The generalised counterpart to the OR-match is *match some*.

$$\mid \text{ match}_{\text{some}} : \text{Module} \times \text{ModQuery} \rightarrow \mathbb{F} \text{ModAdapt}$$

To implement *match_{some}* a similar accumulation strategy to that used for *match_{all}* can be employed. As with *match_{all}*, at each stage the algorithm will attempt to match a query unit against one of the units in the unit set, removing the unit from the set before proceeding to the next step. However unlike the *match_{all}* algorithm, the *match_{some}* algorithm proceeds to the next step regardless of whether a match was found between the current query and module units. The accumulated adaptation passed to the next level is formed by both merging the adaptations representing matches between the two current units with the accumulated value, and by leaving the accumulated value unchanged.

8.3.3.3 ONE-match

The method *match one* corresponds to the XOR match strategy given in the previous section.

$$\mid \text{ match}_{\text{one}} : \text{Module} \times \text{ModQuery} \rightarrow \mathbb{F} \text{ModAdapt}$$

The implementation for *match_{one}* is an extension of the implementation for *match_{xor}*; by considering the instantiation sets $is_j = \text{match}_{\text{unit}}(u, q_j)$, for each q_j in the query set and for each module unit u . Adaptations of the form $(i, \{u\})$ are generated, where i belongs to one of the instantiation sets is_j . The complete set of instantiations is generated by considering all units in the module.

8.3.3.4 A hybrid match strategy

A fourth kind of search strategy is a generalisation of the first three, where the user can specify what unit queries they want to match using a very basic “combination logic” syntax. This logic has three combinators; *and*, *or* and *xor*, similar to the three two-unit query matching strategies.

$$\begin{aligned} \text{CombLogic} ::= & \text{and}\langle\langle \text{CombLogic} \times \text{CombLogic} \rangle\rangle \\ & | \text{or}\langle\langle \text{CombLogic} \times \text{CombLogic} \rangle\rangle \\ & | \text{xor}\langle\langle \text{CombLogic} \times \text{CombLogic} \rangle\rangle \\ & | \text{unit}\langle\langle \text{UnitName} \rangle\rangle \end{aligned}$$

The function $\text{match}_{\text{hybrid}}$ takes a query combination, a query, a module and returns a set of module adaptations.

$$| \text{match}_{\text{hybrid}} : \text{Module} \times \text{ModQuery} \times \text{CombLogic} \rightarrow \mathbb{F} \text{ModAdapt}$$

The above function firstly normalises the query combination into a disjunctive normal form, represented by a set of literal sets. For example the query combination, applied to the queries $\{a, b, c, d\}$:

$$(a \text{ and } b) \text{ or } (c \text{ xor } d)$$

is normalised to

$$\{\{a, b, c\}, \{a, b, d\}, \{a, b\}, \{c\}, \{d\}\}.$$

The next step in the algorithm is to apply an ALL-match to each of the query sets generated in the first step. Finally the result is formed by taking the union of all individual results.

Note that the previous three search methods can easily be constructed with this query language, i.e.:

$$\begin{aligned} \text{all}\{q_1, \dots, q_m\} &= q_1 \text{ and } q_2 \dots \text{ and } q_m \\ \text{some}\{q_1, \dots, q_m\} &= q_1 \text{ or } q_2 \dots \text{ or } q_m \\ \text{one}\{q_1, \dots, q_m\} &= q_1 \text{ xor } q_2 \dots \text{ xor } q_m \end{aligned}$$

Novice users of the search tool mightn't necessarily use this combination language. However more experienced users may find it valuable for constructing queries

which are more complex than those achievable with the other three search methods. Furthermore, application specific front-end search tools which build on this search tool will require the extra flexibility that is afforded by this combination language (see Chapter 9 for more details).

8.4 Matching templates

In this section ideas presented in the previous section are applied to matching templates in CARE. Unit queries in this case are fragment and type specifications, as well as operator declarations and assertions. The unit matching algorithms used are those described in Chapter 7. The *unitsOf* function in this case returns the set of fragments, types, declarations and the assertions (these include definitions, axioms, theorems and applicability conditions) which appear in the template. The module adaptation type *ModAdapt* becomes the template adaptation type (*TemplateAdapt*) defined in Section 4.4.

Example 8.1 Consider the search query, consisting of fragment and type specifications, together with an operator declaration and two assertions, given in Fig. 8.3. Supposing the ALL match strategy is nominated, in which case a template matches the query if it contains units matching each of the unit queries. The accumulator template, given in Fig. C.1 on page 271 of Appendix C, is one possible candidate. The query types `Nat` and `List` are matched against the template types `Element` and `List` respectively. The fragments `rev`, `append` and `nil` match the template fragments `processList`, `processElem` and `base` respectively. The declaration of `rev` matches the declaration of the parameter `f`. The first assertion for `rev` matches the first applicability condition of the accumulator template, while the second assertion matches the second applicability condition. The other two applicability conditions, while not matched must still be included as part of the closed subset. They are included as partially instantiated proof obligations and will need to be proved, along with the other two conditions. The resulting template adaptation is given in Fig. 8.4.

Type `Nat` has specification: \mathbb{N} .
 Type `List` has specification: $\text{seq } \mathbb{N}$.

Fragment `reverse(s>List)` has specification:
 output `r>List` such that $r = \text{rev } s$.

Fragment `append(e:Nat,s>List)` has specification:
 output `r>List` such that $r = \text{append}(e, s)$.

Fragment `nil()` has specification:
 output `r>List` such that $r = \langle \rangle$.

Theory definition of `rev` :
 $\text{rev} : \text{seq } \mathbb{N} \rightarrow \text{seq } \mathbb{N}$;
 $\text{rev}(\langle \rangle) = \langle \rangle$,
 $\forall h : \mathbb{N}; t : \text{seq } \mathbb{N} \bullet \text{rev}(\text{append}(h, t)) = \text{rev}(t) \frown \langle h \rangle$.

Figure 8.3: A search query for matching templates

Adapt Accumulator with:
 $E \rightsquigarrow \mathbb{N}, \text{Acc} \rightsquigarrow \text{seq } \mathbb{N}, f(a) \rightsquigarrow \text{rev}(a)$,
 $hd(a, b) \rightsquigarrow \text{append}(b, a), dh(a, b) \rightsquigarrow b \frown \langle a \rangle, \text{base} \rightsquigarrow \langle \rangle$;
 $\text{List} \rightsquigarrow \text{List}$,
 $\text{Element} \rightsquigarrow \text{Nat}$,
 $\text{processList} \rightsquigarrow \text{reverse}$,
 $\text{processElem} \rightsquigarrow \text{append}$,
 $\text{base} \rightsquigarrow \text{nil}$
 include `processList` with inputs $x \mapsto s$ and outputs $y \mapsto r$

Figure 8.4: A match with the reverse query

8.5 A generic search engine

This section describes a generic search engine which forms the basis for building the prototype retrieval tool in CARE.

8.5.1 Interactive matching

Section 6.7 described interactive matching at the expression level. This idea can be extended to the module level. In a basic model for searching the library, a query is given and a search is done for all those modules which match the query in some

sense. Once the search process is completed, a set of adaptations, representing possible matches, is returned.

Consider two different scenarios: in the first scenario the user is concerned with finding a single solution to their query. In this case it is desirable that having found a solution, the search engine queries the user to see if it is appropriate, and if so terminates the search. Potential savings in time can be made by not having to do an exhaustive search.

The second scenario is where the user wants to find all solutions to a particular query (or indeed they may want to check whether there are any solutions). Here interaction with the user isn't required; instead the user is presented with a set of solutions after an exhaustive check has been completed.

The search engine has four separate levels of interaction: none, guidance only, interaction after each module, and interaction after each match.

$$\begin{array}{l} \textit{InteractionLevel} ::= \textit{automated} \\ \quad \quad \quad \quad \quad | \textit{guidance_only} \\ \quad \quad \quad \quad \quad | \textit{interaction_after_module} \\ \quad \quad \quad \quad \quad | \textit{interaction_after_match} \end{array}$$

The *automated* level simply represents the case where there is no interaction, with results being summarised at the end of the process. As mentioned this level is useful when the user wants to generate all of the matches, for a given search query, or to determine whether or not any modules match a particular query.

The next level, *guidance only*, gives the user feedback on the state of the searching process, outputting matches to the screen as they are found. While this level doesn't allow the user to jump out of the search process, it does give the user a better idea on how the search is progressing. However this verbose mode may tend to slow the search to a certain degree.

The third level *interaction after module*, summarises the matches after each module has been searched. The user is able to view these matches, and can decide whether to continue the process or to terminate the process with one of these matches. This method is useful when the user is interested in finding a module which matches their query, as opposed to all modules matching their query.

The final level *interaction after match* gives the user the option of continuing or quitting after *each* match is found. In this way it is even finer than the previous method. This method will be useful in the case where each module is returning large numbers of matches.

8.5.2 User Configurable Options

The prototype search engine allows the choice of two different equivalences at the expression level: alpha-equivalence and AC-equivalence.

$$\begin{aligned} \textit{ExprEquiv} ::= & \text{alpha_equivalence} \\ & | \text{AC_equivalence} \end{aligned}$$

The search engine allows the user to enable or disable type-constrained matching.

$$\textit{TypeCheck} ::= \text{enable} \mid \text{disable}$$

The search engine also allows a choice between the four module matching strategies described in Section 8.3.

$$\begin{aligned} \textit{Strategy} ::= & \text{all} && - \text{match all} \\ & | \text{some} && - \text{match some} \\ & | \text{one} && - \text{match exactly one} \\ & | \text{hybrid}\langle\langle \textit{CombLogic} \rangle\rangle \end{aligned}$$

8.5.3 A generic search engine

A collection of modules (or *library*) is modelled as a set of modules. No assumptions are made about the structure of the library.

$$\textit{Library} == \mathbb{F} \textit{Module}$$

The search engine takes as input a module query, a library, an interaction level, a module matching strategy, an expression equivalence and a type-checking switch. The search engine calls the appropriate module matching strategies. It is assumed here that these algorithms have been extended to handle interaction and type-checking. A set of module adaptations are returned.

$$\left| \begin{array}{l} \text{search} : \text{ModQuery} \times \text{Library} \times \text{Strategy} \times \text{InteractionLevel} \\ \quad \times \text{ExprEquiv} \times \text{TypeCheck} \rightarrow \mathbb{F} \text{ModAdapt} \end{array} \right.$$

The inputs and outputs of the search engine are fairly raw. As such the search engine will usually be used by building front-end and back-end support tools which gather and process the inputs and outputs.

8.6 Discussion

The approach to matching modules described in this chapter is general, and is applicable to a wide variety of formal languages. Indeed “instantiating” the approach to a particular formal language involves implementing the *unitsOf* function, and developing a framework for adapting the individual units in the language.

Because of the flat structure of *templates*, implementation of the *unitsOf* function in CARE is relatively simple. The problem is more difficult for languages which allow modules to include other modules. One way of implementing *unitsOf* in this case is to “flatten out” the module, returning all units in the module, as well as any units in included modules. However this approach is not necessarily optimal.

Different strategies could be developed for matching against such hierarchically structured modules. One strategy might be to search included modules only to a particular depth. Another strategy would be to search the parent module first, and if this fails, then search the child modules. Finally, by instantiating *unitsOf* appropriately the approach can be applied to hierarchically structured modules.

The approach to module matching described in this chapter is more general than the approach described by Zaremski and Wing [97]. Firstly, the approach described here allows for a more general unit adaptation framework (beyond just parameter instantiation). Secondly the scope of the kind of units which can appear in modules is extended beyond functions. Thirdly, the Zaremski and Wing approach is restricted to the ALL-match strategy.