

Chapter 4

A Structured Approach to Adaptation

4.1 Introduction

The previous chapter contained a general proposal for a framework which supports adaptation of software components for reuse. In this chapter the ideas presented in Chapter 3 are illustrated in depth by application to the CARE language.

In chapter 2 the CARE language was partitioned into three separate levels of constructs - expressions, units and modules. For each level of constructs, adaptation techniques will be defined in this chapter.

Adaptation techniques for expressions, units and modules are described separately in Sections 4.2, 4.3 and 4.4 respectively. For expressions, the technique explored is instantiation of formal parameters. Instantiation is explained and formal definitions are given for each of the different expressions.

For units, the techniques described are renaming of textual parameters, renaming of unit arguments and reordering of unit I/O arguments. Each of the techniques are described separately, and formal definitions of operations for applying the techniques are sketched. In general the definitions for these operations are straightforward, but tedious, so rather than giving full definitions, the operations are simply illustrated on examples in many cases. However it is noted that the definitions have been

carried through to implementation as part of the prototype tools which have been integrated with the CARE toolset.

For modules, the techniques described are subsetting (as defined in the previous chapter) and adapting target code structures to achieve parametric polymorphism at the code level. Both of these techniques are described informally and illustrated by examples. A definition of a function is sketched which incorporates these module-adapting techniques, as well as integrates the techniques for adapting the underlying expressions and units.

In Section 4.5 an example development is presented which illustrates the different adaptation techniques described in this chapter.

4.2 Adapting expressions

The mathematical expressions used in CARE can include parameters ranging over sets, functions and relations (see Section 2.2). Occurrences of parameters within an expression can be substituted by other expressions, thus giving a means of adapting components at the expression level. In the following section the substitution process, otherwise known as *parameter instantiation*, is described. Parameter instantiation was part of the existing CARE tools, however this section formalises the approach for the first time. Note that some of the low-level functions have not been formalised here, instead only their signatures are given, with examples illustrating their functionality.

4.2.1 Formal parameter instantiations

Expressions are *instantiated* by replacing occurrences of parameters by other non-parametric expressions. Where all parameters in an expression are replaced the expression is said to have been *fully* instantiated, otherwise the expression is said to be *partially* instantiated.

To describe how parameters in an expression are to be replaced, a *formal parameter instantiation* is given. The instantiation is essentially a finite partial mapping

from parameters to expressions containing placeholders such that:

- function parameters are mapped to terms,
- set parameters are mapped to sets,
- relation parameters are mapped to formulae.

The mappings are finite because there are only ever finitely many parameters to instantiate. The mappings are partial indicating that not all parameters need to be instantiated. An instantiation of a parameter p to an expression e is represented by the notation $p \rightsquigarrow e$.

<p><i>Inst</i></p> <p>$finsts : FunctionParam \rightsquigarrow Term$</p> <p>$rinsts : RelationParam \rightsquigarrow Fmla$</p> <p>$sinsts : SetParam \rightsquigarrow Set$</p>

Each function parameter is instantiated to a term; this term may contain placeholders which refer to the arguments of the parameter. Similarly relation instantiations may also contain placeholders. The term $ph(n)$ representing the n th placeholder will be written $\langle n \rangle$ here. For example the instantiation $f \rightsquigarrow \langle \langle 2 \rangle \rangle \frown \langle 1 \rangle$, means instantiate the parameter f with the term formed by concatenating the singleton list containing the second argument of f with the first argument of f . The concrete syntax for the above expression is $f(a, b) \rightsquigarrow \langle b \rangle \frown a$, where a and b are arbitrary placeholder symbols, such that they are distinct from any variable and function symbols.

4.2.2 Alpha-equivalence

Intuitively, two mathematical expressions are alpha-equivalent ($=_\alpha$) if they are the same up to renaming of bound variables. The notion of alpha-equivalence will be used in Chapter 5 and onwards to define matches between two expressions.

$$\left| \begin{array}{l} \hline _ =_{\alpha} _ : Term \leftrightarrow Term \\ \hline \forall t_1, t_2 : Term \bullet t_1 =_{\alpha} t_2 \Leftrightarrow \\ (t_1 = \text{var } x \wedge t_2 = \text{var } x) \vee \\ (t_1 = \text{ph}(j) \wedge t_2 = \text{ph}(j)) \vee \\ (t_1 = \text{fnApplic}(f, as_1) \wedge t_2 = \text{fnApplic}(f, as_2) \wedge as_1 =_{\alpha} as_2) \vee \\ (t_1 = \text{termparam}(f, as_1) \wedge t_2 = \text{termparam}(f, as_2) \wedge as_1 =_{\alpha} as_2) \end{array} \right.$$

Two sequences of terms are alpha-equivalent if they are the same length, and the corresponding terms are alpha-equivalent.

$$\left| \begin{array}{l} \hline _ =_{\alpha} _ : \text{seq } Term \leftrightarrow \text{seq } Term \\ \hline \forall ts_1, ts_2 : \text{seq } Term \bullet ts_1 =_{\alpha} ts_2 \Leftrightarrow \\ (\#ts_1 = \#ts_2 \wedge \forall j \in \text{dom } ts_1 \bullet ts_1(j) =_{\alpha} ts_2(j)) \end{array} \right.$$

The definition of alpha-equivalence for formulae is defined below.

$$\left| \begin{array}{l} \hline _ =_{\alpha} _ : Fmla \leftrightarrow Fmla \\ \hline \forall f_1, f_2 : Fmla \bullet f_1 =_{\alpha} f_2 \Leftrightarrow \\ (f_1 = \text{true} \wedge f_2 = \text{true}) \vee \\ (f_1 = \text{not}(f_3) \wedge f_2 = \text{not}(f_4) \wedge f_3 =_{\alpha} f_4) \vee \\ (f_1 = \text{binaryConn}(b, f_3, f_4) \wedge f_2 = \text{binaryConn}(b, f_5, f_6) \\ \wedge f_3 =_{\alpha} f_5 \wedge f_4 =_{\alpha} f_6) \vee \\ (f_1 = \text{relation}(r, as_1) \wedge f_2 = \text{relation}(r, as_2) \wedge as_1 =_{\alpha} as_2) \vee \\ (f_1 = \text{paramrelation}(r, as_1) \wedge f_2 = \text{paramrelation}(r, as_2) \wedge as_1 =_{\alpha} as_2) \vee \\ (f_1 = \text{quant}(q, mvs_1, f_3) \wedge f_2 = \text{quant}(q, mvs_2, f_4) \wedge \#mvs_1 = \#mvs_2 \wedge \\ f_4 =_{\alpha} \text{renameVars}(f_3, \{j \in \text{dom } mvs_1 \bullet \pi_1 mvs_1(j) \mapsto \pi_1 mvs_2(j)\})) \end{array} \right.$$

Example 4.1 The formulae $\forall x : X; y : Y \bullet f(x, y) \wedge P(z)$ and $\forall a : X; b : Y \bullet f(a, b) \wedge P(z)$ are alpha-equivalent. However $\forall x : X; y : Y \bullet f(x, y) \wedge P(c)$ is not alpha-equivalent to either of the two formulae. \square

4.2.3 Preliminary Definitions

Given an expression, the *free variables* are the set of variables which appear unbound (i.e. not quantified) in the expression. For terms *freeVars* has the following signature:

$$\left| \text{freeVars} : Term \rightarrow \mathbb{F} Var \right.$$

Example 4.2 Supposing that x , y and z are variables. Then the expression $x > y - z$ has free variables $\{x, y, z\}$. In contrast the expression $\forall x : X, z : Z \bullet x > y - z$ has a single free variable $\{y\}$ since in this case x and z are bound by the universal quantifier. \square

freeVars can be extended to instantiations by returning the union of the free variables in the expressions appearing in the three instantiation maps.

$$\mid \text{freeVars} : \text{Inst} \rightarrow \mathbb{F} \text{Var}$$

Note that the function *freeVars* has been overloaded.

Example 4.3 The instantiation $\{f(a, b) \rightsquigarrow a \wedge (b \wedge x), R(a, b) \rightsquigarrow b \leq (a + y) \text{div } z\}$ has free variables x (from the instantiation of f), y and z (from the instantiation of R). \square

The set of variables that appear in a mathematical variable declaration is returned by the function *varSet*.

$$\mid \text{varSet} : \text{MathVarDecls} \rightarrow \mathbb{F} \text{Vars}$$

For a mathematical variable declaration the *signature* is defined to be the sequence of sets corresponding to the sets of each variable, in the order given:

$$\mid \text{sig} : \text{MathVarDecls} \rightarrow \text{seq Set}$$

For example given the variable declaration list $x : X, y : Y, z : Z$, then *varSet* returns the set $\{x, y, z\}$ and *sig* returns the list $\langle X, Y, Z \rangle$.

renameVars renames the variables in a given expression in accordance with a given mapping. Note that this may entail renaming bound variables (“ α -conversion”) to avoid capture. For terms:

$$\mid \text{renameVars} : \text{Term} \times (\text{Var} \leftrightarrow \text{Var}) \rightarrow \text{Term}$$

Example 4.4 The *renameVars* operation maps the expression $\forall x : X \bullet P(x, y) \wedge R(z)$ and renaming map $\{x \mapsto s, y \mapsto t, z \mapsto u\}$ to the expression $\forall s : X \bullet P(s, t) \wedge R(u)$ \square

Similar functions are required for the other kinds of expressions.

The function *substPlaceHolders* replaces placeholders by the corresponding expression from a list, i.e. the placeholder $\langle n \rangle$ is replaced by the n th term in the list of terms:

$$\mid \text{substPlaceHolders} : \text{Term} \times (\text{seq Term}) \rightarrow \text{Term}$$

Note that this function may require α -conversion to avoid capture.

Example 4.5 Given an expression $g(\langle 2 \rangle, \langle 1 \rangle + \langle 2 \rangle)$ and an expression list $\langle h(x, y), \#s \rangle$. Then *substPlaceHolders* maps these to the expression $g(\#s, h(x, y) + \#s)$. \square

4.2.4 Instantiating sets

Instantiation of sets is straightforward, “given sets” remain unchanged, set constructors are instantiated by instantiating their sub-expressions, while parametric sets are instantiated by a substitution of the parameter by a set.

$$\begin{array}{l} \overline{\text{instantiate} : \text{Set} \times \text{Inst} \rightarrow \text{Set}} \\ \text{instantiate}(\text{given } s, i) = \text{given } s \\ \text{instantiate}(\text{constructor}(c, ss), i) = \text{constructor}(c, \text{instantiate}(ss, i)) \\ \text{instantiate}(\text{paramset } s, i) = i.\text{sinsts}(s) \end{array}$$

$$\begin{array}{l} \overline{\text{instantiate} : \text{seq Set} \times \text{Inst} \rightarrow \text{seq Set}} \\ \text{instantiate}(ss, i) = \{j : 1 \dots \#ss \bullet j \mapsto \text{instantiate}(ss(j), i)\} \end{array}$$

4.2.5 Instantiating terms

Terms are instantiated by considering placeholders and variables, non-parametric function applications and parametric function applications separately:

- placeholders¹ and unbound variables are left unchanged;

¹Normally terms which are to be instantiated would not contain placeholders; however this case is included for completeness and will be useful later.

- non-parametric function applications are instantiated by instantiating the argument list;
- suppose $f \rightsquigarrow e$ is part of the instantiation map, then a parametric function of the form $f(a_1, \dots, a_m)$ is instantiated by firstly replacing the function application by e , then replacing any occurrences of placeholders $\langle j \rangle$ where $j \in 1..m$ in e by the term that results from instantiating a_j .

$$\begin{array}{|l}
\hline
\textit{instantiate} : \textit{Term} \times \textit{Inst} \rightarrow \textit{Term} \\
\hline
\textit{instantiate}(\textit{ph } n, i) = \textit{ph } n \\
\textit{instantiate}(\textit{var } v, i) = \textit{var } v \\
\textit{instantiate}(\textit{fnapplic}(f, as), i) = \textit{fnapplic}(f, \textit{instantiate}(as, i)) \\
\textit{instantiate}(\textit{termparam}(f, as), i) = \textit{substPlaceHolders}(i.\textit{finsts}(f), m) \\
\textbf{where} \\
m = \{j : 1.. \#as \bullet j \mapsto \textit{instantiate}(as(j), i)\} \\
\hline
\textit{instantiate} : \textit{seq Term} \times \textit{Inst} \rightarrow \textit{seq Term} \\
\hline
\textit{instantiate}(es, i) = \{j : 1.. \#es \bullet j \mapsto \textit{instantiate}(es(j), i)\}
\end{array}$$

Example 4.6 Suppose the parameters f, g, h are defined, then given an instantiation i , of the above parameters, where:

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

The term $t_1 == g(h(p, x + y))$ is instantiated as follows:

$$\begin{aligned}
&\textit{instantiate}(t_1, i) \\
&= \textit{instantiate}(g(h(p, x + y)), i) \\
&= w(\textit{instantiate}(h(p, x + y), i)) \\
&= w(x + y)
\end{aligned}$$

The term $t_2 == f(h(m, x), g(x))$ is instantiated as follows:

$$\begin{aligned}
&\textit{instantiate}(t_2, i) \\
&= \textit{instantiate}(f(h(m, x), g(x)), i) \\
&= u(\Delta_2, v(\Delta_1, \Delta_2))
\end{aligned}$$

where Δ_1 is the result of instantiating the first argument of the call to f , and Δ_2 is the result of instantiating the second argument of the call to f , i.e.:

$$\begin{aligned}
\Delta_1 &= \textit{instantiate}(h(m, x), i) = x \\
\Delta_2 &= \textit{instantiate}(g(x), i) = w(x)
\end{aligned}$$

Therefore the result of instantiation the term t_2 by i is:

$$\text{instantiate}(t_2, i) = u(w(x), v(x, w(x))).\square$$

4.2.6 Instantiating formulae

Formulae are instantiated by considering the different cases as follows:

- truth, negations, binary connectives and non-parametric relation applications are instantiated by instantiating their sub-expressions;
- parametric relation applications are instantiated in much the same manner as their function counterparts;
- quantified formulae are instantiated as follows:
 - first, to avoid capture of free variables in instantiations, introduce a new variable list whose signature is equal to the instantiated signature of the original;
 - rename the variables in the formula;
 - instantiate the formula as per usual.

instantiate : $Fmla \times Inst \rightarrow Fmla$

instantiate(true, i) = true

instantiate(not f , i) = not *instantiate*(f , i)

instantiate(binaryConn(b , f_1 , f_2), i) =
binaryConn(b , *instantiate*(f_1 , i), *instantiate*(f_2 , i))

instantiate(relation(r , es), i) = relation(r , *instantiate*(es , i))

instantiate(paramrelation(r , es), i) = *substPlaceHolders*($i.rinsts$ (r), m)

where

$m = \{j : 1 .. \#es \bullet j \mapsto \text{instantiate}(as(j), i)\}$

instantiate(quant(q , vs , f), i) = quant(q , vs_1 , f_1)

where $\exists vs_1 \bullet sig(vs_1) = sig(\text{instantiate}(vs, i)) \wedge$

(*varSet*(vs_1) \cap *freeVars*(i)) = $\emptyset \wedge$

$f_1 = \text{instantiate}(\text{renameVars}(f, \{j : 1 .. \#vs \bullet \text{first } vs(j) \mapsto \text{first } vs_1(j)\}), i)$

A variable declaration list is instantiated by instantiating each of the sets in the list.

$$\left| \begin{array}{l} \textit{instantiate} : \textit{VarDeclar}s \times \textit{Inst} \rightarrow \textit{VarDeclar}s \\ \textit{instantiate}(\langle \rangle, i) = \langle \rangle \\ \textit{instantiate}(\langle (v, T) \rangle \wedge vs, i) = \\ \quad \langle (v, \textit{instantiate}(T, i)) \rangle \wedge \textit{instantiate}(vs, i) \end{array} \right.$$

Example 4.7 Suppose the parameters R , P and f are defined with the following signatures:

$$R : X \times Y, P : X \times Y \times Z, f : X \times Y \rightarrow Y$$

Given an instantiation i of the above parameters:

$$i == \{R(a, b) \rightsquigarrow b < a, P(a, b, c) \rightsquigarrow s(a, b) \leq c, f(a, b) \rightsquigarrow b + t(x, z)\}$$

Then the formula $f_1 == R(x + y, 0)$ is instantiated as follows:

$$\textit{instantiate}(f_1, i) = \textit{instantiate}(R(x + y, 0), i) = 0 < x + y$$

To instantiate the formula, f_2 , where:

$$f_2 == \forall x : X, y : Y, z : Z \bullet R(x, f(x, y)) \wedge P(x, y, z),$$

note that the variables x and z appear as local variables in the formula f_2 , as part of the universal quantifier, as well as appearing freely in the instantiation of the parameter f . The first step in instantiating f_2 is to rename the local variables x and z to u and v (say) resulting in the formula :

$$\forall u : X, y : Y, v : Z \bullet R(u, f(u, y)) \wedge P(u, y, v).$$

Next f_2 is instantiated as follows:

$$\begin{aligned} & \textit{instantiate}(f_2, i) \\ &= \forall u : X, y : Y, v : Z \bullet \textit{instantiate}(R(u, f(u, y)) \wedge P(u, y, v), i) \\ &= \forall u : X, y : Y, v : Z \bullet \Delta_1 < u \wedge s(u, y) \leq v \end{aligned}$$

where

$$\Delta_1 = \textit{instantiate}(f(u, y), i) = y + t(x, z)$$

Therefore

$$\textit{instantiate}(f_2, i) = \forall u : X, y : Y, v : Z \bullet y + t(x, z) < u \wedge s(u, y) \leq v. \square$$

4.2.7 Correctness preservation

Instantiation replaces formal parameters with actual values, in effect replacing abstract constructs by more concrete ones. If a parameterised component can be shown to be correct without assumptions being made about the values the parameters can take, then it follows that this correctness of the component is maintained regardless of the values passed to the parameters.

In some instances, to show correctness of the parameterised component constraints must be placed on the range of values that the parameters can take. These constraints are referred to as *applicability conditions*. To show that correctness is preserved, the instantiated applicability conditions become proof obligations, that must be discharged by the user.

4.3 Adapting units

This section describes techniques for adapting CARE units. Formal definitions for these techniques are sketched, however since these definitions are straightforward but tedious, full definitions have been omitted. In CARE, units are categorised as either types, fragments, operator declarations or assertions. Each of these units consists of one or more mathematical expressions, which can contain parameters; therefore units inherit parameter instantiation as one possible adaptation technique. In the following section a number of other ways of adapting units are considered including:

- renaming of unit identifiers;
- renaming of input and/or output variables;
- reordering of input and/or output variables.

Note that the main focus of this section will be on adapting fragments, since they cover all of the three techniques listed above. Adaptation techniques for other kinds of units in CARE could be defined along similar lines to those for fragments, except that adaptation of these other units is restricted to instantiation of parameters and renaming identifiers.

4.3.1 Renaming identifiers

Each unit has a unique identifier which enables the unit to be referenced from within other units as well as by other tools/processes. Note that the name used to identify a unit is not important provided it is consistent throughout the scope of the unit's use.

With this in mind observe that identifiers can be renamed to achieve meaningful naming within the user's application domain. To ensure that the correctness of the program is preserved, units must be renamed at the point of definition as well as anywhere that the unit is referenced. Also the identifiers must be renamed to new identifiers which do not already appear within the scope of the renaming. As well as unit names, identifiers can include the names of reports used in branching fragments.

$$Identifier == UnitName \cup Report$$

To rename the identifiers within a unit, a mapping from existing identifiers to new identifiers could be defined. The mapping is an injective map, where identifiers can be mapped to themselves.

$$Renaming == Identifier \mapsto Identifier$$

Example 4.8 The renaming mapping

$$\{X \rightsquigarrow A, Y \rightsquigarrow B, \mathbf{append} \rightsquigarrow \mathbf{cons}, \mathbf{head} \rightsquigarrow \mathbf{first}\}$$

renames the types X and Y to A and B respectively, and the fragments \mathbf{append} and \mathbf{head} to \mathbf{cons} and \mathbf{first} . \square

Given a renaming mapping, a simple fragment could be adapted by renaming the fragment name at the point of definition, as well as applying the renaming to any types in the input and output declaration lists, and finally by applying the renaming to any occurrences of unit identifiers within the implementation part of the fragment.

$$\mid \text{rename} : SimpleFragment \times Renaming \mapsto SimpleFragment$$

Similar renaming functions could be defined for the other kinds of units (i.e. branching fragments, types, operator declarations and assertions). To rename the

identifiers within a collection of units (such as in a module), the renaming functions must be applied to all units to ensure that the identifiers are renamed consistently.

$$\mid \text{ rename} : \text{Template} \times \text{Renaming} \rightarrow \text{Template}$$

4.3.2 Renaming input and output variables

The number and types of input and outputs for a fragment are described by a variable declaration. The names of these declared variables can be changed without changing the overall meaning of the fragment, provided the changes are done in a consistent manner throughout the unit. The following section describes the technique of renaming input and output variables of fragments; these ideas can be applied to any function-like unit which takes inputs and returns outputs.

To rename the input and output variables of a fragment, a *variable renaming* would be defined; this is an injective mapping from the original set of variables to a new set of variables.

$$\text{VarRenaming} == \text{Var} \mapsto \text{Var}$$

To ensure correctness is preserved, the variables in the range of the variable renaming must be disjoint from those appearing in the input or output lists.

Renaming of the input and output variables of a fragment can lead to possible clashes with local variables which occur either as bound variables in quantified formulae within the specification, or in the binding lists within the implementation part. Prior to applying the variable renaming map to avoid name clashes, any local variables in the fragment that appear in the range of the variable mapping would be given new names. Having eliminated any potential clashes it is then a simple task to rename the I/O variables in accordance with the renaming mapping. Variables would be renamed where they are introduced, as well as at any points where they are used within the fragment. The function *renameVars* has the following signature.

$$\mid \text{renameVars} : \text{Fragment} \times \text{VarRenaming} \rightarrow \text{Fragment}$$

The definition of this operation (which might be given separately for simple and branching fragments) is straightforward, however since it is tedious further details have been omitted. The use of this function is illustrated on an example below.

```

Fragment gimli(x:X,y:Y) has
specification:
  precondition  $\forall a : A \bullet P(a, x) \wedge Q(x, y)$ 
  output r:R,s:S such that  $f(r, s) = g(x, y)$ 
implementation:
  assign gribble(x,y) to b:B;
  assign crok(b,x) to r:R,s:S;
  return r,s.

```

Figure 4.1: The fragment gimli prior to variable renaming

Example 4.9 There are two main steps to rename the input and output variables of the fragment `gimli`, given in Fig. 4.1, under the variable renaming mapping $\{x \mapsto a, y \mapsto b, r \mapsto m, s \mapsto n\}$. Firstly the local variables must be renamed to avoid name clashes. In this example, the variable a appears in both the universal quantifier in the precondition and in the renaming map, therefore must be renamed to avoid capture. Similarly the local variable b appears in both the implementation part and the renaming, therefore is also renamed. The renaming is arbitrary provided it doesn't lead to further name clashes. In this case let us suppose a and b are renamed to the variables c and d respectively.

Having eliminated any potential name clashes, the renaming mapping can be applied to the fragment (with the above renamings added). The result of applying the renaming mapping is given in Fig. 4.2. \square

```

Fragment gimli(a:X,b:Y) has
specification:
  precondition  $\forall c : A \bullet P(c, a) \wedge Q(a, b)$ 
  output m:R,n:S such that  $f(m, n) = g(a, b)$ 
implementation:
  assign gribble(a,b) to d:B;
  assign crok(d,a) to m:R,n:S;
  return m,n.

```

Figure 4.2: The fragment gimli after variable renaming has been performed

4.3.3 Reordering input and output variables

Similar to the idea of renaming the input and output variables of a fragment is that of reordering of variables. Reordering of the arguments of fragments is however more complicated in that not only must the variables be reordered at the point where they are introduced, but also wherever the fragment is called. This means that the reordering must be applied to any other fragment which calls the fragment in question.

The reordering of variables is described by a *permutation map*, which is a finite partial injective map from the naturals to the naturals, such that the domain and range are equal, given by some contiguous subset of the naturals beginning at 1.

$$Permutation == \{m : \mathbb{N} \rightsquigarrow \mathbb{N} \mid \exists n : \mathbb{N} \bullet \text{dom } m = \text{ran } m = 1 \dots n\}$$

Permutations are represented as a set of ordered pairs; e.g. the permutation $p = \{1 \mapsto 3, 2 \mapsto 1, 3 \mapsto 2\}$ means map the first variable to the third position, the second to the first position and the third to the second position.

A function *permute*, which reorders a variable declaration list is required.

$$\mid \text{permute} : \text{VarDecls} \times \text{Permutation} \rightarrow \text{VarDecls}$$

For example, the declaration list $x:X, y:Y, z:Z$ is mapped by the permutation map p above, to $y:Y, z:Z, x:X$.

Reordering of inputs and outputs for a simple fragment is described via a mapping for the input variables and another mapping for the output variables. Reordering of inputs and outputs for a branching fragment is described by a permutation map for the inputs and a permutation map for the outputs in each branch of the specification. I/O permutation maps could be modelled for both simple and branching fragments as follows.

$$IOPerm == \text{Permutation} \times \text{seq}_1 \text{Permutation}$$

The first element in this pair describes the permutation for the input, while the second element describes the permutation(s) for the outputs in each of the branches in the specification. Note that for simple fragments, the second element in the pair would always consist of a single permutation.

Given a variable permutation map, to reorder the input and output variables of a fragment there would be two steps. Firstly the variables must be reordered at the point of introduction. Secondly the reordering must be applied wherever the fragment is called within an implementation (including possibly within the same fragment's implementation, in the case of recursive calls).

Reordering the input and output variables within the fragment's specification is straightforward. It is simply a matter of applying the appropriate permutation map to the inputs and to each of the output lists. For simple fragments, the operation *permuteSSpec*, would be defined. For branching fragments, the operation *permuteBSpec*, would be defined. These operations have the following signatures.

$$\left| \begin{array}{l} \textit{permuteSSpec} : \textit{SimpleFragmentSpec} \times \textit{IOPerm} \rightarrow \textit{SimpleFragmentSpec} \\ \textit{permuteBSpec} : \textit{BFragmentSpec} \times \textit{IOPerm} \rightarrow \textit{BFragmentSpec} \end{array} \right.$$

Given an input/output reordering for a given fragment, then wherever the fragment is called in an implementation (whether this is within another fragment, or the same fragment in the case of a recursive call), the reordering must be applied accordingly. More precisely given a call to a fragment for which an input/output reordering is given, then the input permutation must be applied to the arguments of the fragment call, and the output permutation(s) must be applied to the local variables that the result(s) are bound to.

$$\left| \textit{permuteFragCalls} : \textit{FragBody} \times (\textit{FragmentName} \rightarrow \textit{IOPerm}) \rightarrow \textit{FragBody} \right.$$

The function *permuteVars* which permutes the variables in a fragment is defined below in terms of the functions for permuting variables in the specification and body.

$$\left| \begin{array}{l} \textit{permuteVars} : \textit{Fragment} \times (\textit{FragmentName} \rightarrow \textit{IOPerm}) \rightarrow \textit{Fragment} \\ \hline \textit{permuteVars}(\textit{simple } F, \textit{iomap}) = \textit{simple } F' \\ \textbf{where} \\ \quad F'.\textit{name} = F.\textit{name} \\ \quad F'.\textit{spec} = \textit{permuteSSpec}(F.\textit{spec}, \textit{iomap}(F.\textit{name})) \\ \quad F'.\textit{body} = \textit{permuteFragCalls}(F.\textit{body}, \textit{iomap}) \\ \textit{permuteVars}(\textit{branching } F, \textit{iomap}) = \textit{branching } F' \\ \textbf{where} \\ \quad F'.\textit{name} = F.\textit{name} \\ \quad F'.\textit{spec} = \textit{permuteBSpec}(F.\textit{spec}, \textit{iomap}(F.\textit{name})) \\ \quad F'.\textit{body} = \textit{permuteFragCalls}(F.\textit{body}, \textit{iomap}) \end{array} \right.$$

```

Branching fragment frodo(x:X,y:Y,z:Z) has
specification:
  precondition P(x,y,z)
  if G(x,y,z) then report yes
    with output u:U,v:V such that Q1(x,y,z,u,v)
  else report no with output w:W such that Q2(x,y,z,w)

Fragment pippin(a:U,b:V) has
specification:
  output c:U,d:Z,e:X such that R(a,b,c,d,e).

Fragment merry(x:X,y:Y,z:Z) has
implementation:
  cases frodo(x,y,z) of:
    yes: assign output to p:U,q:V;
         assign pippin(p,q) to a:U,b:Z,c:X;
         return a,b,c
    no:  assign output to s:W;
         return s.

```

Figure 4.3: Fragment collection prior to reordering of input/output

Example 4.10 Fig. 4.3 contains specifications of the fragments `frodo` and `pippin`, as well as the implemented fragment `merry`, which is implemented in terms of the two previous fragments. Suppose reordering mappings are defined for the inputs and each of the outputs for the fragments `pippin` and `frodo` as follows:

```

reorder(pippin, in) = {1 ↦ 2, 2 ↦ 1}
reorder(pippin, out) = {1 ↦ 2, 2 ↦ 3, 3 ↦ 1}
reorder(frodo, in) = {1 ↦ 3, 2 ↦ 1, 3 ↦ 2}
reorder(frodo, out branch 1) = {1 ↦ 2, 2 ↦ 1}
reorder(frodo, out branch 2) = {1 ↦ 1}

```

Then the reordering mappings can be applied to the collection of fragments by firstly reordering the input and output variables of the specified-only fragments `frodo` and `pippin`, and then applying the reorderings to the fragment calls in the implementation of `merry`. The result of applying these reorderings is given in Fig 4.4.

□

Some syntactic sugar Given a variable list $\langle x : X, y : Y, z : Z \rangle$, a variable renaming map $\{x \mapsto a, y \mapsto b, z \mapsto c\}$, and a variable permutation map $\{1 \mapsto$

```

Branching fragment frodo(y:Y,z:Z,x:X) has
specification:
  precondition P(x, y, z)
  if G(x, y, z) then report yes
    with output v:V,u:U such that Q1(x, y, z, u, v)
  else report no with output w:W such that Q2(x, y, z, w)

Fragment pippin(b:V,a:U) has
specification:
  output e:X,c:U,d:Z such that R(a, b, c, d, e).

Fragment merry(x:X,y:Y,z:Z) has
implementation:
  cases frodo(y,z,x) of:
    yes: assign output to q:V,p:U
          assign pippin(q,p) to c:X,a:U,b:Z;
          return a,b,c
    no:  assign output to s:W;
          return s.

```

Figure 4.4: Fragment collection after reordering of input/output variables

$3, 2 \mapsto 2, 3 \mapsto 1$ (i.e. swap the first and third variables), then the result of applying these two maps to the original variable list is the variable list $\langle c : Z, b : Y, a : X \rangle$.

For convenience these two maps can be expressed as $\langle z \mapsto c, y \mapsto b, x \mapsto a \rangle$. That is the renaming and reordering are expressed as a single statement. It is represented by a list of pairs, the first element in each pair refers to an original variable in the list, the second element refers to what the variable will be renamed to. Given this statement together with the variable list, it is a relatively straightforward process to infer the original two mappings.

Remark Renaming and reordering are kept separate in the abstract syntax for better modularity, and also because the scope of renaming is at the unit level, whereas the scope of reordering is at the module level.

4.3.4 The integrated approach

The *instantiate* function given in Section 4.2 can be extended to the unit level by applying the *instantiate* function to each of the mathematical expressions in the

unit. For example, instantiating a fragment involves instantiating the individual mathematical expressions which make up the fragment's specification, i.e. the precondition, guards (where applicable), and postcondition(s).

$$\mid \textit{instantiate} : \textit{Fragment} \times \textit{Inst} \rightarrow \textit{Fragment}$$

Fragments can be adapted by combining the three techniques described in this section with instantiation of formal parameters. A *fragment adaptor* consists of an instantiation, an identifier renaming map, a variable renaming map and a variable reordering map which may apply to the fragment's I/O as well as any of the fragments called in the implementation:

$$\textit{FragmentAdapt} == \textit{Inst} \times \textit{Renaming} \times \textit{VarRenaming} \times (\textit{FragmentName} \rightarrow \textit{IOPerm})$$

Any or all of the individual parts of a fragment adaptation may be omitted by the user, in which case default values are given. The default value for the instantiation part is the *trivial* instantiation, while the default value for each of the other parts is the empty set. Given a fragment adaptation then a fragment is adapted by applying each of the four adaptation techniques in turn. The function *adapt_fragment* which performs adaptations of simple fragments is defined below. A similar function for adapting branching fragments could also be defined along the same lines.

$$\left| \begin{array}{l} \textit{adapt_fragment} : \textit{Fragment} \times \textit{FragmentAdapt} \rightarrow \textit{Fragment} \\ \hline \textit{adapt_fragment}(F, (i, r, vr, iomap)) = \\ \quad \textit{permuteVars}(\textit{rename}(\textit{renameVars}(\textit{instantiate}(F, i), vr), r), iomap) \end{array} \right.$$

Type specifications simply consist of mathematical expressions, so types can be adapted by renaming identifiers and instantiating formal parameters. A *type adaptor* consists of an instantiation and a renaming:

$$\textit{TypeAdapt} == \textit{Inst} \times \textit{Renaming}$$

The function *adapt_type* performs a type adaptation:

$$\mid \textit{adapt_type} : \textit{Type} \times \textit{TypeAdapt} \rightarrow \textit{Type}$$

Assertions and operator declarations can also be adapted by instantiating parameters and renaming identifiers.

4.3.5 Correctness preservation

Since unit identifiers act merely as pointers to the unit, renaming of identifiers is correctness preserving provided that the identifiers are renamed consistently throughout the scope of the corresponding unit's use, and provided the renaming does not introduce clashes with existing unit identifiers.

Renaming of the input and output arguments of fragments will be correctness preserving provided the variables are renamed consistently throughout the entire fragment, and the renaming does not introduce any clashes with existing I/O variables. Clashes with other local variables are avoided by doing a preprocess renaming of bound variables prior to applying the renaming of I/O arguments.

Reordering of the I/O arguments of a fragment is correctness preserving provided the arguments of any call to this fragment are similarly reordered.

4.4 Adapting modules

The final level of components for which adaptation techniques are developed are modules. Being more or less a collection of units, modules will inherit the adaptations described in the previous two sections. In this section the adaptations specific to modules are developed; in particular selecting subsets of modules and adapting target language constructs. Recall that in CARE modules are referred to as templates.

4.4.1 Selecting subsets of templates

Recall that the body of a template consists of a set of formally specified units (such as fragments, types and theories). In some cases some of these units are optional, and therefore it is possible to include self-contained subsets of templates. As an example consider the `Natural numbers` template (see Fig. C.5). It contains a number of different primitives for manipulating naturals, however in general not all of these will be required to solve a certain problem. So it should be possible to include certain subsets of the primitive components.

In selecting a subset of a template (referred to earlier as *subsetting*), to ensure correctness is preserved the subset must be self-contained. That is, any component *used* in the subset should also be *contained* in the subset. For a unit its *supporters* are defined to be the set of units which are used by the unit:

$$\left| \text{supporters} : \text{Unit} \rightarrow \mathbb{F} \text{UnitName} \right.$$

Given the set of units which make up the body of a template (it is assumed that each unit is named uniquely), together with a named subset (a set of unit names) of the template body, then the *closure* of the named subset is the set of components formed by including all of the named units, together with any supporters of the named units:

$$\left| \begin{array}{l} \text{closure} : \mathbb{F} \text{UnitName} \times \mathbb{F} \text{Unit} \leftrightarrow \mathbb{F} \text{Unit} \\ \hline \forall m : \mathbb{F} \text{Unit} \bullet \text{closure}(\emptyset, m) = \emptyset \\ \forall m : \mathbb{F} \text{Unit}; x : \text{UnitName} \bullet \\ \quad \exists c : \text{Unit} \bullet c \in m \wedge c.\text{name} = x \Rightarrow \\ \quad \quad \text{closure}(\{x\}, m) = \{c\} \cup \text{closure}(\text{supporters}(c), m) \\ \forall a, b : \mathbb{F} \text{UnitName}; m : \mathbb{F} \text{Unit} \bullet \\ \quad \text{closure}(a \cup b, m) = \text{closure}(a, m) \cup \text{closure}(b, m) \end{array} \right.$$

Example 4.11 Given a template with components $\{A, B, C, D, E, F\}$, where the supporter sets for each component are given as follows:

$$\text{supporters}(A) = \{B, D, E\}$$

$$\text{supporters}(B) = \{D, E\}$$

$$\text{supporters}(C) = \{A, F\}$$

$$\text{supporters}(D) = \emptyset$$

$$\text{supporters}(E) = \emptyset$$

$$\text{supporters}(F) = \{D, E\}$$

Then the closure of the named set $\{A, D\}$ can be calculated as follows:

$$\begin{aligned} \text{closure}(\{A, D\}) &= \text{closure}(\{A\}) \cup \text{closure}(\{D\}) \\ &= \{A\} \cup \text{closure}(\text{supporters}(A)) \cup \{D\} \cup \emptyset \\ &= \{A\} \cup \text{closure}(\{B, D, E\}) \cup \{D\} \end{aligned}$$

$$\begin{aligned}
&= \{A\} \cup \text{closure}(\{B\}) \cup \text{closure}(\{D\}) \cup \text{closure}(\{E\}) \cup \{D\} \\
&= \{A\} \cup \{B\} \cup \text{closure}(\{D, E\}) \cup \{D\} \cup \{E\} \\
&= \{A, B, D, E\} \square
\end{aligned}$$

4.4.2 Adapting target code structures

This section describes a technique for adapting the target code constructs contained within the primitive units of a template. The adaptations explained so far have dealt with the specification part of units, and so it is relatively easy to check that the correctness of adapted units is maintained. However correctness of the implementations of primitive units is outside of the scope of the CARE method, with other complimentary testing/verification methods needing to be applied to such units off-line before such templates are added to the library.

In adapting target code constructs in a particular unit, it is important not to compromise the correctness of the resulting units. Therefore ad hoc approaches to adapting the target code must be avoided. The focus shall be on techniques which do not compromise the correctness of resulting units. One such method described here enables polymorphic target language data structures to be defined.

Suppose the user wants to implement a CARE type to represent a list of natural numbers. Fig. 4.5 shows one possible implementation in C, as a linked list of unsigned integers (the implementation part gives a directive to the code synthesiser tool describing what code is generated for a list of this type - the exact details of what appears in the implementation part are not particularly important within the context of this thesis). However the problems with this approach are twofold. Firstly the user might decide to implement naturals as signed integers (for example) instead; in which case type mismatches will result in the generated code. Secondly with this approach, a template for each of a possibly infinite number of different kinds of lists would be required.

To solve the first problem the CARE type name is linked with the target language data structure name by introducing an *associated code* part. The associated code of the type implementation is added to the resulting target code program. In this case a

```

Type List has
specification: seq N
implementation:
variable l defined by "struct linked_list {unsigned int val;
    struct linked_list * next;}; *l".

```

Figure 4.5: A list of naturals implemented in terms of a linked list of unsigned integers

`typedef` is used to link the CARE type name and the target language data structure name. Fig. 4.6 shows how the CARE type `Nat` can be linked to the target language type of the same name by using a `typedef`. In this case `Nat` is implemented by an unsigned integer. The notation `<|Nat|>` is a directive to the adaptation tool stating that if the CARE type `Nat` is renamed, then this string should also be renamed. As a result after adaptation the name of the target code type will always be consistent with the name of the CARE type.

```

Type Nat has
specification: N
implementation:
variable m defined by "<|Nat|> m"
with associated code
"typedef unsigned int <|Nat|>".

```

Figure 4.6: Implementing natural numbers as unsigned integers

The approach is extended to allow for polymorphic data structures to be implemented at the template level. An example of a list of elements of (mathematical) type E is given in Fig. 4.7. The list is parameterised by linking the underlying list elements, to the CARE type `Elem`. For example, suppose the user required a list of natural numbers, then they might rename `Elem` \rightsquigarrow `Nat` and instantiate $E \rightsquigarrow \mathbb{N}$ as well as perhaps renaming `List` \rightsquigarrow `NatList`. The result of applying such an adaptation is given in Fig. 4.8.

Parametric polymorphism is achieved at the code level in a correctness preserving manner by linking a target code data structure with an identifier in the specification. The target code is adapted by giving an identifier renaming. The approach can be generalised to other target code adaptations in which part of the code is linked with

```

Type Elem has specification: E

Type List has
specification: seq  $\mathbb{N}$ 
implementation:
variable l defined by "<|List|> l"
with associated code
"struct linked_list {<|Elem|> val;
  struct linked_list * next;};",
"typedef struct linked_list POINTER;",
"typedef POINTER * <|List|>".

```

Figure 4.7: A polymorphic linked list

```

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

Type NatList has
specification: seq  $\mathbb{N}$ 
implementation:
variable l defined by "NatList l"
with associated code
"struct linked_list {Nat val; struct linked_list * next;};",
"typedef struct linked_list POINTER;",
"typedef POINTER * NatList;".

```

Figure 4.8: An adaptation of a polymorphic linked list

part of the interface, and is thus adapted by adapting the interface. For example the values of an enumerated type at the target code level might be linked with the values of a mathematical set at the interface level.

Note that while this approach works for the applications we have looked at with C as the target language, we are not claiming here that it will work in all instances; for example lists whose elements are lists which can only be implemented by lazy evaluation. Such considerations are outside of the scope of this thesis, instead the intention here is only to give an insight into one possible solution to target code adaptation.

4.4.3 The integrated approach

One way of adapting a template is to adapt the individual units contained in the template. Since parameter instantiations and identifier renamings are applied to the entire template, a single data structure *UnitAdapt* (referred to as a *unit adaptor*), is used to represent adaptations of a set of the units in the template. *UnitAdapt* is an amalgamation of the different kinds of unit adaptations described in Section 4.3. It is used to apply a formal parameter instantiation, and identifier renaming across the entire template, and variable renamings and reordering to individual fragments.

$ \begin{array}{l} \textit{UnitAdapt} \\ \textit{paraminst} : \textit{Inst} \\ \textit{identifier_renaming} : \textit{Renaming} \\ \textit{io_renaming} : \textit{FragmentName} \rightarrow \textit{VarRenaming} \\ \textit{io_permutation} : \textit{FragmentName} \rightarrow \textit{IOPerm} \end{array} $
--

Individual units can be adapted by calling the function *adapt_unit*, which could be defined by calling one of the unit adapting functions defined in Section 4.3.4.

| $\textit{adapt_unit} : \textit{Unit} \times \textit{UnitAdapt} \rightarrow \textit{Unit}$

A template can also be adapted by *subsetting*, which is described by nominating a subset of the units in the template. Overall a template adaptation is described by a unit adaptor and a unit name set. The unit name set represents the input to the *closure* function defined on page 90, so in particular is not necessarily closed with respect to the template being adapted.

$$\textit{TemplateAdapt} == \textit{UnitAdapt} \times \mathbb{F} \textit{UnitName}$$

The function *adapt_template* applies an adaptation to a template and returns a set of units.

| $\textit{adapt_template} : \textit{Template} \times \textit{TemplateAdapt} \rightarrow \mathbb{F} \textit{Unit}$

4.4.4 Correctness preservation

A template is correct in CARE if each of the units used in the module is at least specified within the template, and each of the non-primitive implemented units is

correct with respect to its specification. Therefore the correctness of a template subset follows from the correctness of the entire module, provided that the subset is self-contained.

Since the correctness of primitive units (i.e. those implemented directly by target code constructs) is outside of the scope of CARE, adaptations to target code must be very conservative. The method proposed here for achieving parametric polymorphism at the code level involves linking the target code data structure with a CARE identifier. Therefore the target code is in effect only changed by applying identifier renaming, which as already argued, is correctness preserving.

4.5 Example - Summing the elements of a list

In this section the adaptation techniques developed in the previous sections are illustrated by development of a CARE program for summing the elements of a list of numbers.

4.5.1 Initial specification

An initial specification for calculating the sum of a list of numbers is given in Fig. 4.9. Included are types for representing natural numbers and lists of natural numbers. Also included is the fragment `sumList` which when implemented should return the sum of a list of natural numbers. Finally the function *sumList* is defined, which provides a mathematical description of summing a list.

4.5.2 Implementing `sumList`

One way of calculating the sum of a list of numbers is to employ an accumulation strategy. The elements of the list can be processed one at a time, their value being added to an accumulated sum. The process continues until the entire list has been processed in this manner. Initially the accumulated value is zero. To implement `sumList`, the `Associative Commutative Accumulator` template (see

```

Type Nat has specification:  $\mathbb{N}$ 
Type NatList has specification:  $\text{seq } \mathbb{N}$ 

Fragment sumList(s:NatList) has specification:
output r:Nat such that  $r = \text{sumList}(s)$ 

Theory definition of sumList:
sumList :  $\text{seq } \mathbb{N} \rightarrow \mathbb{N}$ ;
sumList( $\langle \rangle$ ) = 0,
 $\forall h : \mathbb{N}; t : \text{seq } \mathbb{N} \bullet \text{sumList}(\text{append}(h, t)) = h + \text{sumList}(t)$ 

```

Figure 4.9: Initial specification for summing a list of numbers

Fig. C.2), which is a version of the accumulator template in which the step function is associative commutative is adapted.

A description of how the accumulator template is to be adapted is given in Fig. 4.10. Firstly the formal parameters are instantiated - all the template's parameters are given a value, so the instantiation is complete. Next some of the template identifiers are renamed to be consistent with the initial specification, and to be appropriate for the application domain. Finally the input and output variables of the fragment `processList` are renamed to be consistent with those of `sumList`.

```

Adapt Associative Commutative Accumulator with:
  E  $\rightsquigarrow \mathbb{N}$ , f(a)  $\rightsquigarrow \text{sumList}(a)$ , hd(a, b)  $\rightsquigarrow a + b$ , base  $\rightsquigarrow 0$ ;
  List  $\rightsquigarrow \text{NatList}$ ,
  Element  $\rightsquigarrow \text{Nat}$ ,
  processList  $\rightsquigarrow \text{sumList}$ ,
  accumulator  $\rightsquigarrow \text{sumListAcc}$ ,
  processElem  $\rightsquigarrow \text{add}$ ,
  base  $\rightsquigarrow \text{zero}$ 
  include processList with inputs  $x \mapsto s, y \mapsto r$ 

```

Figure 4.10: Adapting the accumulator template

The result of applying this template adaptation is given in Fig. 4.11. The fragment `sumList` is refined by `processList`, while the fragments `sumListAcc`, `one`, `zero` and `decomposeList` are also added to the development.

Also as a result of adapting the template, the applicability conditions are instan-

```

Fragment sumList(s:NatList) has
implementation:
  return sumListAcc(s,zero).

Fragment sumListAcc(x:NatList,y:Nat) has
specification: output z:Nat such that  $z = fold(x, y)$ 
implementation:
  cases decomposeList(x) of:
    nonempty: assign output to v:Nat,w:NatList;
              assign add(v,y) to y:Nat;
              return sumListAcc(w,y)
    empty:    return y

Fragment add(e:Nat,a:Nat) has
specification:
  output r:Nat such that  $r = e + a$ .

Fragment zero() has
specification: output b:Nat such that  $b = 0$ .

Branching fragment decomposeList(x:NatList) has
specification:
  if  $x \neq \langle \rangle$  then report nonempty
  with output h:Nat,t:NatList such that  $x = append(h, t)$ 
  else report empty.

```

Figure 4.11: Implementing `sumlist` using the accumulator template

tiated to give the following proof obligations which must be satisfied to ensure the correctness of the resulting unit set:

$$\begin{aligned}
 & sumList(\langle \rangle) = 0 \\
 & \forall h : \mathbb{N}, t : seq \mathbb{N} \bullet sumList(append(h, t)) = h + sumList(t) \\
 & \forall x, y : \mathbb{N} \bullet x + y = y + x \\
 & \forall x, y, z : \mathbb{N} \bullet x + (y + z) = (x + y) + z
 \end{aligned}$$

The first two conditions follow directly from the definition of `sumList`, while the other two conditions follow simply from the commutativity and associativity laws for naturals under addition.

4.5.3 Implementing natural number primitives

The next step in the program development is implementing the units `Nat`, `add` and `zero`. Suppose the user chooses to use the `Natural numbers` template (see Fig. C.5), which implements natural number primitive in terms of unsigned integers. An adaptation of this template is given in Fig. 4.12. The type name `Num` is renamed to `Nat`. From the template the user only includes the units `Num`, `add` and `zero`; since this is a self-contained unit set, no other units need to be included. Finally the input and output variables of `add`, as well as the output of `zero` are renamed to match the units in Fig. 4.11. The resulting implementations for `Nat`, `add` and `zero` are given in Fig. 4.13.

```
Adapt Natural numbers with:
  Num ~> Nat
  include Num,
  include add with inputs x ↦ e, y ↦ a and outputs z ↦ r,
  include zero with outputs n ↦ b.
```

Figure 4.12: Adapting the unsigned integers template

```
Type Nat has
specification: N
implementation:
  variable i defined by "Nat i"
with associated code
"typedef unsigned int Nat;".

Fragment add(e:Num,a:Num) has
implementation:
  { target code elided. }

Fragment zero() has
implementation:
  { target code elided. }
```

Figure 4.13: Implementing natural number primitives using unsigned integers

4.5.4 Implementing list primitives

The final step in the development is to provide implementations for the units `NatList` and `decomposeList`. Suppose the user selects the `Linked lists` template (see Fig. C.4), which contains implementations for a number of list primitives in terms of a linked list data structure.

A description of how this template will be adapted is given in Fig. 4.14. Firstly the formal parameter E is instantiated to the set of natural numbers. Next a number of units are renamed to fit with the current development. From the template a subset consisting of the template units `List`, `Element` and `apndlI` is nominated. Finally the input variable of `apndlI` is renamed such that it is consistent with the input variable of `decomposeList`.

```

Adapt LinkedLists with:
   $E \rightsquigarrow \mathbb{N}$ ;
  List  $\rightsquigarrow$  NatList
  Element  $\rightsquigarrow$  Nat
  apndlI  $\rightsquigarrow$  decomposeList
  include apndlI with inputs  $v \mapsto x$ 
  include Element
  include List

```

Figure 4.14: Adapting the linked lists template

Note that the inclusion set is not self-contained, since `apndlI` depends on `hdI`, `tlI` and `isEmpty` - these units must also be included. The result of applying the adaptation is given in Fig. 4.15.

4.6 Discussion

In this chapter a number of techniques for adapting components have been developed. As a result a single component has the potential to solve many problems. The approach unifies the various techniques into a single framework, which includes instantiation of formal parameters, renaming of identifiers, renaming and reordering

Type `NatList` has
specification: `seq(N)`
implementation:
 variable `l` defined by `"NatList l;"`
with associated code
`"struct linked_list { Nat val; struct linked_list * next;};"`,
`"typedef struct linked_list POINTER;"`,
`"typedef POINTER * NatList;"`.

Branching fragment `decomposeList(x:NatList)` has
implementation:
 cases `isEmpty(v)` of:
 yes: report empty
 no: assign `hdl(v)` to `h:Nat`;
 assign `tll(v)` to `t:NatList`;
 report nonempty and return `h,t`

Branching fragment `isEmpty(v:NatList)` has
specification:
 result defined by cases
 if $v = \langle \rangle$ then report yes else report no
implementation:
 { target code elided }

Fragment `hdl(v:NatList)` has
specification:
 precondition $v \neq \langle \rangle$
 output `e0:Nat` such that $e0 = head\ v$
implementation:
 { target code elided }

Fragment `tll(v:NatList)` has
specification:
 precondition $v \neq \langle \rangle$
 output `l2:NatList` such that $l2 = tail\ v$
implementation:
 { target code elided }

Figure 4.15: An adaptation of the linked list template

variables, inclusion of a subset of a template and adapting target code constructs. Some of these techniques are new, while others have been used before in isolation.

Instantiating formal parameters is a commonly used adaptation technique. Identifier renaming can be thought of as a similar technique to parameter instantiation, however identifiers can only be instantiated to other identifiers. Reordering of the arguments of units is quite different, and to the best of the author's knowledge has not appeared in any of the reuse literature. Similarly, while module subsetting is discussed in the context of component matching by Zaremski and Wing [95], it does not appear in the framework of adaptation; consequently the issue of ensuring the subset is self-contained is not raised. Harwood [33] proposes a slightly different approach to module subsetting, where a module is included initially as a minimal subset, with other module units included only as required. However the approach proposed here is more general, in that it allows the user to include the minimal subset or a larger subset as required, thus allowing far greater flexibility. Finally the idea of adapting components by changing underlying target-code data structures is similar to Volpano and Kieburtz's [91] approach, however the approach described in this thesis is more general and could be extended to include other kinds of target code adaptation.

Each of the techniques is illustrated within the context of the CARE language, however they are all relatively general, and are applicable to a wide range of other languages.

A large part of the design and development of templates in CARE is establishing the correctness of the implementation against the specification (e.g. see [60] for a proof of correctness for the accumulator template). It is therefore important that any techniques for adapting components in CARE are *correctness preserving*.

The ideas presented in this chapter are an extension of the template adaptation framework developed earlier in the CARE project [40]. In particular the first two techniques described (parameter instantiation and renaming) had already been developed; they are presented here for completeness. Note that this work is partly inspired by an earlier version of a template adaptation tool, named the Finder tool

[16].

The Finder tool takes a “Finder construct” as entered by the user and uses it as a key into the library. Finder constructs come in six different forms - refinements, scenarios, target language data refinements, data refinements, theories and implementation bodies. Each Finder construct contains a list of arguments, describing how the corresponding library component could be adapted. The exact form of these arguments is not clear from the documentation, but in essence it includes parameter instantiations, identifier renaming and subsetting information.

The approach taken here is quite different to that used by the Finder tool. In particular the approach here leads to a more intuitive and simpler interface for reusable components, with the user provided with more assistance in adapting components.

All of the techniques described in this chapter have been implemented and integrated with the existing CARE tools. The author was primarily responsible for the design and implementation of target language adaptation, argument reordering and subsetting and co-designer and implementor of all of the other techniques.