A Formal Approach to Component Adaptation and Composition

David Hemer

School of Computer Science
University of Adelaide
Outline

- Project aims
- Background
  - Programs
  - Library components
- Retrieval
- Adaptation
Project aims

- Support component-based software engineering (CBSE) of high-integrity software
- Goal is to make savings in verification effort
- Build on existing refinement calculus work
- Define reusable library components
- Develop methods and tools for
  - Locating suitable components
  - Adapting them to meet the user’s specific needs
Programs

- Programs developed using stepwise refinement
- Unimplemented requirements represented by specifications

```plaintext
addelem(in e:E, in s:List, out r:List)
  pre isNonRep(s) ∧ len(s) ≤ 50
  post ran(r) = ran(s) ∪ {e} ∧ len(r) ≤ 50 ∧ isNonRep(r).
```

- Specifications replaced with lower level code (refinement)
Library components

• Encapsulate refinements between specifications and implementations
• Proofs of correctness done off-line by design expert
• Module = collection of refinements specific to particular data structure or algorithm
• Template = parameterised (higher-order) collection of refinements used to solve general problems
Component retrieval

- *Match* program specification (query) against library component specifications
- Replace program specification by corresponding implementation part(s) of library component
- Search for *exact* or *relaxed* matches
- Matches defined as (library) parameter instantiations
Exact matching

A query $Q$ and library component specification $S$ are an exact match w.r.t. a parameter instantiation $\pi$ iff

1. $Q_{\text{pre}} \Leftrightarrow \pi S_{\text{pre}}$

2. $Q_{\text{post}} \Leftrightarrow \pi S_{\text{post}}$

Match signatures to align I/O variables and types
Adapting library components

- Library components often need to be adapted to satisfy a program specification
- Individual library component can be modified, e.g.
  - parameter instantiation
  - textual renamings
  - adding, dropping and reordering of I/O arguments
- Library components can be used in combination with other components, e.g.,
  - sequential composition
  - case analysis
Adaptation techniques

- Some modification techniques are built into the matching algorithms, e.g. parameter instantiation, textual renamings
- Define modification techniques and architectures for combining components using templates
- Templates are parameterised so are general and applicable in a variety of circumstances
- In general satisfying a program specification will involve matching against a sequence of adaptation templates, until a match can be found with module
- We aim to semi-automate adaptation using search tactics
Example

- Find an implementation for `sum_lengths`

  fragment `sum_lengths` (in `x, y:L`, out `z:N`)
  post `z = len(x) + len(y)`.

- Use library components `add` and `length`

  fragment `add` (in `x, y:Nat`, out `z:Nat`)
  post `z = x + y`.

  fragment `length` (in `x:List`, out `n:Nat`)
  post `n = len(x)`.

- But first need to decompose the problem

- Apply the FUNDECOMP adaptation template
Functional decomposition

template FUNDECOMP [X;Y;U;V;W; f : U × V → W; g : X × Y → U; h : X × Y → V] is

fragment main (in x:X, in y:Y, out w:W)
pre P(x,y)
post w = f(g(x,y), h(x,y))
::= gfrag(x,y)::u:U;
    hfrag(x,y)::v:V;
    ffrag(u,v).
Functional decomposition (cont.)

fragment \( f\text{frag} \) (in \( u:U \), in \( v:V \), out \( w:W \))
post \( w = f(u, v) \).

fragment \( g\text{frag} \) (in \( x:X \), in \( y:Y \), out \( u:U \))
pre \( P(x, y) \)
post \( u = g(x, y) \).

fragment \( h\text{frag} \) (in \( x:X \), in \( y:Y \), out \( v:V \))
pre \( P(x, y) \)
post \( v = h(x, y) \).

end template.
Applying the template

- Templates are used by instantiating the parameters

\[ X, Y \mapsto \text{seq } \mathbb{N} \]
\[ U, V, Z \mapsto \mathbb{N} \]
\[ f \mapsto \lambda a, b \cdot a + b \]
\[ g \mapsto \lambda a, b \cdot \text{len}(a) \]
\[ h \mapsto \lambda a, b \cdot \text{len}(b) \]
\[ P \mapsto \lambda a, b \cdot \text{true} \]
The result

- `sum_lengths` implemented by `main`
- Three new fragment specifications added to the program
  
  \[
  \text{fragment } ffrag \ (\text{in } u:N, \text{ in } v:N, \text{ out } w:N) \\
  \text{post } w = u + v.
  \]
  
  \[
  \text{fragment } gfrag \ (\text{in } x:L, \text{ in } y:L, \text{ out } u:N) \\
  \text{post } u = \text{len}(x).
  \]
  
  \[
  \text{fragment } hfrag \ (\text{in } x:L, \text{ in } y:L, \text{ out } v:N) \\
  \text{post } v = \text{len}(y).
  \]

- We now need to find implementations for these fragments
2nd development step

- `ffrag` can be implemented directly by library fragment `add`
- `gfrag` and `hfrag` are similar to `length`, but have an extra input argument
- Adapt using a wrapper template to remove unused input argument
Wrapper template

template DROPINPUT \([X; Y; Z; P : X \rightarrow \mathbb{B}; f : X \rightarrow Z]\) is

fragment main\((in \ x : X, in \ y : Y, out \ z : Z)\)
pre \(P(x)\)
post \(z = f(x)\)
:: = frag1\((x)\).

fragment frag1\((in \ x : X, out \ z : Z)\)
pre \(P(x)\)
post \(z = f(x)\).
end template.
Applying DROPINPUT

- Match $gfrag$ against $main$
- Instantiate parameters

$$P \mapsto \text{true}, f \mapsto \lambda \ a \cdot \text{len}(a)$$

- New specified only fragment

  fragment $frag1$ (in $x:L$, out $u:N$)
  post $u = \text{len}(x)$.

- Can be implemented directly by $\text{length}$
- Repeat for $hfrag$
- Development completed
Summary

• Savings in (verification) effort can be made by using pre-verified library components
• Only viable if support for locating and adapting components is available
• Adaptation techniques defined using templates
• Adaptation can be semi-automated by developing search tactics