Transformational Synthesis

Ras Bodik
CS294-2  Software Synthesis
Spring 2006

Motivation, example

- dot product:
  \[
  \text{dot}(x,y,n) = \begin{cases} 
  0 & \text{if } n=0 \\
  \text{dot}(x,y,n-1) + x[n]y[n] & \text{else}
  \end{cases}
  \]

- we want to compute (specification):
  \[
  f(a,b,c,d,n) = \text{dot}(a,b,n) + \text{dot}(c,d,n)
  \]

- synthesis optimizes it into this (implementation):
  \[
  f(a,b,c,d,n) = \begin{cases} 
  0 & \text{if } n=0 \\
  f(a,b,c,d,n-1) + a[n]b[n] + c[n]d[n] & \text{else}
  \end{cases}
  \]

- benefit:
  one recursion (loop) rather than two

Notation

\[
f(x) = \begin{cases} 
  1 & \text{if } x=0 \text{ or } x=1 \\
  f(x-1)+f(x-2) & \text{else}
  \end{cases}
\]

is rewritten as

\[
f(0) = 1
f(1) = 1
f(x+2) = f(x+1)+f(x)
\]

Notation

\[
\text{concat}(x,y) = \begin{cases} 
  y & \text{if } x=\text{nil} \\
  \text{cons(car(x), \text{concat}(\text{cdr(x)}, y))} & \text{else}
  \end{cases}
\]

is rewritten as

\[
\text{concat}(\text{nil}, z) = z
\text{concat}(\text{cons}(x,y), z) = \text{cons}(x,\text{concat}(y,z))
\]

Inference (transformation) rules

1. Definition:
   - introduce a new recursive equation
     - ex.: \(f(a,b,c,d,n) = \text{dot}(a,b,n) + \text{dot}(c,d,n)\)

2. Instantiation:
   - substitute into an existing equation
     - \(f(a,b,c,d,n) = \text{dot}(a,b,n) + \text{dot}(c,d,n)\) becomes
       \(f(a,b,c,d,0) = \text{dot}(a,b,0) + \text{dot}(c,d,0)\)

3. Unfolding:
   - substitution of an equation on the right-hand side
     - given: \(g(x+1) = g(x)+1\)
     - \(f(a) = h(g(y)+1)\) unfolds into \(f(a) = h(g(y)+1)\)
Inference (transformation) rules

4. Folding:
   • the inverse to folding
   • given lhs \( \subseteq \) rhs, replace an instance of rhs with lhs
   • \( f(a) \leftarrow h(g(y)+1) \) is folded with \( g(x+1) \leftarrow g(x)+1 \) into \( f(a) \leftarrow h(g(y+1)) \)

Inference (transformation) rules

5. Abstraction:
   • introduce a where clause
   • \( f(a) \leftarrow h(g(y)+1) \) becomes
     \( f(a) \leftarrow h(g+y) \) where \( (u,v) = (g,y,1) \)

Inference (transformation) rules

6. Laws:
   • rewrite rhs with a law such as associativity

Synthesis strategy

1. make necessary definitions
2. instantiate
3. for each instantiation unfold repeatedly, after each unfold:
   a. apply laws and where-abstraction
   b. fold repeatedly

User involvement:
   • Invention needed in 1, 2.
   • Discretion needed in a.
   • rest is mechanical.

Example 1:

Spec:
- \( \text{fact}(0) \leftarrow 1 \)
- \( \text{fact}(n+1) \leftarrow (n+1)\text{fact}(n) \)
- \( \text{factlist}(0) \leftarrow \text{nil} \)
- \( \text{factlist}(n+1) \leftarrow \text{cons}(\text{fact}(n+1),\text{factlist}(n)) \)

Derivation:
5. \( g(n) \leftarrow (\text{fact}(n+1),\text{factlist}(n)) \)
   def (eureka)
6. \( g(0) \leftarrow (\text{fact}(1),\text{factlist}(0)) \)
   \( \leftarrow (\text{nil}) \)
   instantiate 5 with n=0
   unfold 2, 1, law "*", unfold 4

Example 1, cont’d

7. \( \text{g}(n+1) \leftarrow (\text{fact}(n+2),\text{factlist}(n+1)) \)
   inst. 5 with n=n+1
   \( \leftarrow (\text{fact}(n+1),\text{fact}(n)) \)
   \( \leftarrow (\text{fact}(n+1),\text{factlist}(n)) \)
   un 2,4
   \( \leftarrow (\text{fact}(n+1),\text{cons}(u,v)) \) where \( (u,v) = (\text{fact}(n+1),\text{factlist}(n)) \)
   abstract
   \( \leftarrow (\text{fact}(n+1),\text{cons}(u,v)) \) where \( (u,v) = g(n) \)
   fold with 5
8. \( \text{factlist}(n+1) \leftarrow \text{cons}(\text{fact}(n+1),\text{factlist}(n)) \)
   this is def 4, copied
   \( \leftarrow \text{cons}(u,v) \) where \( (u,v) = (\text{fact}(n+1),\text{factlist}(n)) \)
   abstract
   \( \leftarrow \text{cons}(u,v) \) where \( (u,v) = g(n) \)
   fold with 5
Strategies for applying the transformations

- **Goal:**
  - avoid enumerating all possible transformations
  - by restricting explored transformation sequences
    - it’s still a search
    - still can be called synthesis 😌
- **Interesting questions:**
  - some loss of generality
    - i.e., not complete wrt to given definitions, rewrite rules

Observations

- almost all optimizations are sequences of
  - unfoldings, followed by
    - rewriting by lemmas, followed by
      - foldings
- associativity, commutativity, where-abstraction
  - performed just before folding
  - so, perform only to enable a fold (called forced fold)

Algorithm 1

1. perform an arbitrary unfold or a rewrite
   - repeat, terminating arbitrarily
2. perform an arbitrary forced fold
   - repeat while folds are possible

The prototype

the user enters:

1. equations, including the "eureka" definitions
2. rewriting lemmas
3. list of instantiated left hand sides of equations

the system will start its derivations from (3)

Example interaction with the system

- See Example 1

Folding

- uses a matching routine:
  - given expressions e and e’,
  - find substitution σ that transforms e into e’
- example:
  - e = n+m+k
  - e’ = m+(n+1+k)
  - σ(n) = n+1
**Folding with where-abstraction**

- **Example (Fibonacci):**
  1. \( f(0) \rightleftharpoons 1 \)
  2. \( f(1) \rightleftharpoons 1 \)
  3. \( f(x+2) \rightleftharpoons f(x+1) + f(x) \)
  4. \( g(x) \rightleftharpoons (f(x+1), f(x)) \)
- **now the system instantiates and unfolds**
  5. \( g(x+1) \rightleftharpoons (f(x+1)+f(x), f(x+1)) \)
- **and tries to fold (5) with (4)**
  - components of (4) are available, yielding
  \( g(x+1) \rightleftharpoons (u+v, u) \) where \( (u,v)=(f(x+1), f(x)) \)
  \( \rightleftharpoons (u+v, u) \) where \( (u,v)=g(x) \)

**Future developments (as of 1977)**

- **Automate development of auxiliary functions**
  - i.e., where does \( g(x) \rightleftharpoons (f(x+1), f(x)) \) come from?
- **The problem, again, simplified:**
  - **given a specification** \( f \)
  \( f(x+1) \rightleftharpoons f(x) ... f(x) \)
  - **synthesize** \( g \), a more efficient implementation of \( f \)
  \( g(x+1) \rightleftharpoons g(x) ... g(x) \)
- **More precisely, we want**
  1. allow more general substitutions: \( g(\sigma(x)) \rightleftharpoons g(x) ... g(x) \)
  2. \( f \) to be expressible in terms of \( g \):
  \( f(\sigma(x)) \rightleftharpoons g(x) ... g(x) \)

**Example: factlist**

\[
\begin{align*}
\text{fact}(n+1) & \rightleftharpoons (n+1)\text{fact}(n) \\
\text{factlist}(n+1) & \rightleftharpoons \text{cons}(\text{fact}(n+1), \text{factlist}(n))
\end{align*}
\]

- \( \sigma(n) = n+1 \) relates levels of the tree
- if we choose \( g(n) \rightleftharpoons \{ \text{fact}(n+1), \text{factlist}(n) \} \)
  then \( g(n+1) \) can be expressed in terms of \( g(n) \)