Transformational Synthesis

Ras Bodik
CS294-2 Software Synthesis
Spring 2006

Motivation, example

- dot product:
  dot(x,y,n) ⇐ if *n=0* then *0* else *dot(x,y,n-1) + x[n]y[n]*
- we want to compute (specification):
  f(a,b,c,d,n) ⇐ *dot(a,b,n) + dot(c,d,n)*
- synthesis optimizes it into this (implementation):
  f(a,b,c,d,n) ⇐ if *n=0* then *0*
  else *f(a,b,c,d,n-1) + a[n]b[n] + c[n]d[n]*
- benefit:
  one recursion (loop) rather than two

Notation

f(x) ⇐ if *x=0* or *x=1* then *1* else *f(x-1)+f(x-2)* fi

is rewritten as

f(0) ⇐ 1
f(1) ⇐ 1
f(x+2) ⇐ *f(x+1)+f(x)*

Notation

concat(x,y) ⇐ if *x=nil* then *y*
else cons(car(x), concat(cdr(x), y))

is rewritten as

concat(nil, z) ⇐ *z*
concat(cons(x,y), z) ⇐ *cons(x,concat(y,z))*

Inference (transformation) rules

1. Definition:
   - introduce a new recursive equation
   - ex.: *f(a,b,c,d,n)* ⇐ *dot(a,b,n) + dot(c,d,n)*
2. Instantiation:
   - substitute into an existing equation
   - *f(a,b,c,d,n)* ⇐ *dot(a,b,n) + dot(c,d,n)* becomes
     f(a,b,c,d,0) ⇐ *dot(a,b,0) + dot(c,d,0)*

Inference (transformation) rules

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 ⇐ rhs, replace an instance of rhs with lhs

   - *f(a) ⇐ h(g(y)+1)* is folded with *g(x+1) ⇐ g(x)+1* into *f(a) ⇐ h(g(y+1))*

Inference (transformation) rules

5. Abstraction:
   - introduce a where clause
   - *f(a) ⇐ h(g(y)+1)* becomes *f(a) ⇐ h(u+v)* 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:

- *fact(0)* ⇐ *1*
- *fact(n+1)* ⇐ *(n+1)*fact(n)*
- *factlist(0)* ⇐ *nil*
- *factlist(n+1)* ⇐ *cons(fact(n+1),factlist(n))*

Derivation:

5. g(n) ⇐ ⟨*fact(n+1),factlist(n)*⟩
def (eureka)

6. g(0) ⇐ ⟨*fact(1),factlist(0)*⟩
instantiate 5 with n=0

⇐ ⟨*1,nil*⟩
unfold 2, 1, law "*", unfold 4

Example 1, cont'd

7. g(n+1) ⇐ ⟨*fact(n+2),factlist(n+1)*⟩
inst. 5 with n=n+1

⇐ ⟨(n+2)**fact(n+1), cons(fact(n+1), factlist(n)*⟩
un 2,4

⇐ ⟨(n+2)**u, cons(u,v)* ⟩ where ⟨*u,v*⟩ = ⟨*fact(n+1),factlist(n)*⟩
abstract

⇐ ⟨(n+2)**u, cons(u,v)* ⟩ where ⟨*u,v*⟩ = *g(n)*
fold with 5

8. factlist(n+1) ⇐ *cons(fact(n+1),factlist(n))*
this is def 4, copied

⇐ *cons(u, v)* where ⟨*u,v*⟩ = ⟨*fact(n+1),factlist(n)*⟩
abstract

⇐ *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:

  $e = n+m+k$

  $e' = m+(n+1+k)$

  $\sigma(n) = n+1$

Folding with where-abstraction

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

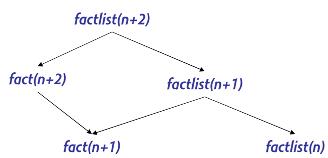
Future developments (as of 1977)

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

Example: factlist

$fact(n+1) \Leftarrow (n+1)*fact(n)$

$factlist(n+1) \Leftarrow cons(fact(n+1), factlist(n))$



- $\sigma(n) = n+1$ relates levels of the tree
  if we choose $g(n) \Leftarrow \langle fact(n+1), factlist(n) \rangle$
  then $g(n+1)$ can be expressed in terms of $g(n)$