Transforming LP Programs

- Accumulators
- Partial Evaluation
- Unfolding
- Folding
Accumulators

Consider the following program.

\[
\text{length1}([], 0).
\]
\[
\text{length1}([H|T], L) :-
\]
\[
\text{length1}(T, L1),
\]
\[
L = L1 + 1.
\]

- The order is forced by \texttt{is}/2

- Can we devise a tail recursive version.
A better length

length2(List, Len) :-
    length2_aux(List, 0, Len).

length2_aux([], Len, Len).
length2_aux([H|T], L1, Len) :-
    L2 is L1 + 1,
    length2_aux(T, L2, Len).

The extra argument is used to pass values from one invocation of length2_aux to the next.
Partial Evaluation

Consider the program:

\[
\begin{align*}
\text{ordered}([],). \\
\text{ordered}([H|[]]). \\
\text{ordered}([H,K|T]) \leftarrow \\
\quad H < K, \\
\quad \text{ordered}([K|T]).
\end{align*}
\]

Suppose that we want to apply this to two element lists.

We could use the predicate ordered, but here is a better idea.
The Evaluation tree of ordered([X, Y]).
The Unfolding

- Identify each derivation that has either been called successfully or truncated.

- Denote the residual calls at the leaves by \textbf{calls}

- $\theta$ the substitution

- The clause recovered for $q$ is $\forall((q \text{ if calls})\theta)$

- Or in this case

  \[
  \text{ordered}([X,Y]) \leftarrow X < Y.
  \]
Unfolding

Suppose that we have a program

\[
A :\ldots, C, \ldots
\]

\[
C :\text{ body-1.}
\]
\[
. \]
\[
. \]
\[
. \]
\[
C :\text{ body-n.}
\]

we can unfold this to
A :- ..., body-1, ...
.
.
.
.
A :- ..., body-n, ...

C :- body-1.
.
.
.
.
C :- body-n.
An example of Unfolding

\[ \text{consecutive_pair}(U,V,[U,V|X]). \]
\[ \text{consecutive_pair}(U,V,[W|Y]) :- \]
\[ \text{consecutive_pair}(U,V,Y). \]

If we do one unfolding step we get new clauses

\[ \text{consecutive_pair}(U,V,[W,U,V|X]). \]
\[ \text{consecutive_pair}(U,V,[W1,W2|Y]) :- \]
\[ \text{consecutive_pair}(U,V,Y). \]

We use these to replace the second clause. Giving:

\[ \text{consecutive_pair}(U,V,[U,V|X]). \]
\[ \text{consecutive_pair}(U,V,[W,U,V|X]). \]
\[ \text{consecutive_pair}(U,V,[W1,W2|Y]) :- \]
\[ \text{consecutive_pair}(U,V,Y). \]
Folding: The model

The general idea is that given clauses:

\[ A :- \ldots \text{ calls, \ldots.} \]

\[ C:- \text{ calls.} \]

We will obtain the folded program

\[ A :- \ldots, C, \ldots \]

\[ C :- \text{ calls.} \]

Actually we need to pay attention to the way that the body of \( C \) is unified with \( C \) in the body of \( A \).
An example using folding and Unfolding

A program that detects pairs \((U,V)\) satisfying \(p(U,V)\) from members of a list \(L\).

\[
\text{pair}(U, V, L) \leftarrow \text{mem}(U, L), \text{mem}(V, L), p(U, V). \quad \% \text{P1}
\]
\[
\text{mem}(U, [U|\_]). \quad \% \text{P2}
\]
\[
\text{mem}(U, [\_V|X]) \leftarrow \text{mem}(U, X). \quad \% \text{P3}
\]

Unfold P1 to obtain

\[
\text{pair}(U, V, [U|X]) \leftarrow \text{mem}(V, [U|X]), p(U, V). \quad \% \text{P5}
\]
\[
\text{pair}(U, V, [W|X]) \leftarrow \text{mem}(U, X), \text{mem}(V, [W|X]), p(U, V). \quad \% \text{P6}
\]

These can replace P1.
The Example Continued

Now unfold P3 and P4 to get:

\[
\begin{align*}
\text{pair}(U,U, [U\mid X]) & : \text{p}(U,U). \quad \% \text{P6} \\
\text{pair}(U,V, [U\mid X]) & : \text{mem}(V,X) \land \text{p}(U,V). \quad \% \text{P7} \\
\text{pair}(U,V, [V\mid X]) & : \text{mem}(U,X) \land \text{p}(U,V). \quad \% \text{P8} \\
\text{pair}(U,V, [W\mid X]) & : \text{mem}(U,X), \text{mem}(V,X), \\
& \quad \text{p}(u,V). \quad \% \text{P9}
\end{align*}
\]

We can fold P9 to obtain:

\[
\text{pair}(U,V, [W\mid X]) : \text{pair}(U,V,X) \quad \% \text{P10}
\]
The transformed program

\[
\begin{align*}
\text{pair}(U, U, [U|X]) & \leftarrow \text{p}(U, U). \quad \text{\% P6} \\
\text{pair}(U, V, [U|X]) & \leftarrow \text{mem}(V, X), \text{p}(U, V). \quad \text{\% P7} \\
\text{pair}(U, V, [V|X]) & \leftarrow \text{mem}(U, X), \text{p}(U, V). \quad \text{\% P8} \\
\text{pair}(U, V, [W|X]) & \leftarrow \text{pair}(U, V, X) \quad \text{\% P10} \\
\text{mem}(U, [U|\_]). & \quad \text{\% P2} \\
\text{mem}(U, [\_V|X]) & \leftarrow \text{mem}(U, X). \quad \text{\% P3}
\end{align*}
\]
Transforming Functional Programs

- Folding and Unfolding again

- Duplicate Computations

- Continuations
Folding and Unfolding

Suppose we have:

let rec length = function [] -> 0
  | _::_:t -> 1 + length t;;

and that we want to define length2 of type
list 'a -> list 'a -> int.

We have the proposition

∀l₁,l₂ : length2l₁ l₂
= (length l₁)+(length l₂)

How can we get a better program than this.
Example continued

If we set $l_1 = []$ we get

$$\text{length} 2[] \ l = (\text{length} [\ ])+(\text{length} \ l)$$
$$= \text{length} \ l$$

Setting $l_1 = a::l$ we get

$$\text{length} 2 \ (a::l) \ l' = (\text{length} \ (a::l))$$
$$+ (\text{length} \ l'))$$
$$= (1 + (\text{length} \ l)) + (\text{length} \ l'))$$
$$= 1 + ((\text{length} \ l')) + (\text{length} \ l))$$
$$= 1 + (\text{length} 2 \ l \ l')$$
Giving the program

let rec length2 =
    function [] -> ( function l -> length l)
    | _:::l1 ->
        function l ->1 + (length2 l1 l);;
Duplicate Computations

let rec fib = function 0 -> 1
| 1 -> 1
| n -> (fib (n-1)) + (fib (n-2));

We use a Eureka insight to define a function
fibpair n = (fib (n + 1)), (fib n)

So

g 0 = 1,1
g (n + 1) = fib (n+1)+ fib(n), fib (n+1)
Even Better

let rec g = function n ->
    if n = 0 then (1,1) else
    let (u,v) = (g (n-1)) in (v, u+v);;
Continuations

A continuation captures the idea of “the rest of the computation”. We can apply it to our old friend reverse. Ie we suppose that we are going to do \((\theta \circ \text{reverse})\) and try to write this as

\[
(\theta \circ \text{reverse}) = r \ l \ \theta
\]

then \text{reverse} \ l = r \ l \ \text{id}

and

\[
\begin{align*}
 r[] \theta &= \theta [] \\
r(a :: l) \theta &= r \ l(\theta \circ (\lambda \ z. z * [a]))
\end{align*}
\]
we can argue

\[
\text{reverse} \quad = \quad r \ l \ id
\]
\[
= \quad r \ l \ (\lambda z. z)
\]
\[
= \quad r \ l \ (\lambda z. z \ * \ [])
\]
\[
= \quad g \ l \ []
\]

\[
g \ [] \ z \quad = \quad r \ [](\lambda z'. z' \ * \ z)
\]
\[
= \quad (\lambda z'. z' \ * \ z)[]
\]
\[
= \quad [] \ * \ z
\]
\[
= \quad z
\]

\[
g \ (a::l) \ z \quad = \quad r(a::l)(\lambda z'. z' \ * \ z)
\]
\[
= \quad r \ l \ (\lambda z'. z' \ * \ z) \circ (\lambda z'. z' \ * \ [a])
\]
\[
= \quad r \ (\lambda z'. z' \ * \ [a]) \ * \ z
\]
\[
= \quad r \ l \ (\lambda z'. z' \ * \ (a :: z))
\]
\[
= \quad g \ l \ (a::z)
\]
let rec rev = function [] -> (function l -> l)
| h::t ->
  function l1 -> (rev t (h::l1));

let reverse t = rev t [];;