Optimization Strategies for FL

Alex Aiken

October 5, 1988

1 A Basic Strategy

Optimizing "housekeeping" operations means removing the data structures that are passed between composed functions. For example, in a program $f \circ g$, a data structure is created by g which is read by f. We wish to eliminate the need for the allocation, writing, and reading of this structure.

This problem has been extensively studied by Phil Wadler. The strategy proposed here is an adaption of a strategy for a first-order functional language based on the lambda calculus.[1]

In the following, f, g, r, and q are functions and p is a predicate. We consider only first-order functions built from composition, construction, conditional, and first-order primitives. It is assumed that function definitions are expanded as necessary and that expressions are without parentheses. All expressions are assumed to be *Safe*.

$$T[\![f \circ (p \rightarrow q; r)]\!] \Rightarrow T[\![p \rightarrow f \circ q; f \circ r]\!] \quad (1)$$

$$T[\![[f_1, \ldots, f_n] \circ g]\!] \Rightarrow T[\![[f_1 \circ g, \ldots, f_n \circ g]]\!] \quad (2)$$
if 1 does not apply

$$T[\![(p \rightarrow q; r) \circ f]\!] \Rightarrow T[\![p \circ f \rightarrow q \circ f; r \circ f]\!] \quad (3)$$
if 1 does not apply

$$T[\![[f_1, \ldots, f_n]]\!] \Rightarrow [T[\![f_1]\!], \ldots, T[\![f_n]\!]] \quad (4)$$

$$T[\![p \rightarrow q; r]\!] \Rightarrow T[\![p]\!] \rightarrow T[\![q]\!]; T[\![r]\!] \quad (5)$$

$$T[\![f_1 \circ \ldots \circ f_n]\!] \Rightarrow T[\![f_1 \circ T[\![f_2 \circ \ldots \circ f_n]]\!]]\!] \quad (6)$$
for $n > 2$

[1] "Deforestation: Transforming Programs to Eliminate Trees", Wadler, ESOP '88.

These rules preserve the semantics of a program. Rules 1-3 are correct algebraic transformations. Rules 4-6 apply the transformations recursively. As shown below, these rules eliminate compositions from a program. Rules 1-3 change the primary connective from a composition of two functions to either a conditional or construction.

Lemma 1.1 Let F be the set of all programs constructable from the primitive function id and the combining forms composition, construction, and conditional. Any $g \in F$ is transformed into an equivalent program without intermediate results (without compositions) using T and three rules about id:

$$T[\![f \circ \mathrm{id}]\!] \Rightarrow T[\![f]\!] \quad (7)$$

$$T[\![\mathrm{id} \circ f]\!] \Rightarrow T[\![f]\!] \quad (8)$$

$$T[\![\mathrm{id}]\!] \Rightarrow \mathrm{id} \quad (9)$$

Proof: [sketch] At any point in the application of the rules, let the "transformed portion" of g be that part which is not enclosed by T. We show by induction on the number of steps in the transformation process that the transformed portion of g has no compositions.

Initially, the transformed portion of the program is empty, so the hypothesis is trivially satisfied. Assuming the hypothesis, applying any of rules 1-6 satisfies the requirements of the induction. If none of 1-6 applies, then it must be because the term is of the form $T[\![f \circ h]\!]$ where either f or h is a primitive function. But the only primitive is id, so rule 7 or 8 must apply. □

2 An Example with Recursion

Of course, FL has many more primitives than id. Additional rules are required that describe properties of primitive functions. The following rules, along with rules 7 and 8, eliminate unnecessary computation:

$$T[\![\mathrm{s}_i \circ [f_1, \ldots, f_n]]\!] \Rightarrow T[\![f_i]\!] \quad \text{if } n \geq i \quad (10)$$

$$T[\![p \rightarrow (p \rightarrow q; r); g]\!] \Rightarrow T[\![p \rightarrow q; g]\!] \quad (11)$$

Rules 7-11 are applied in preference to rules 1-6 if possible. Finally, two rules are required to describe what happens when nothing can be done:

$$T[\![f]\!] \Rightarrow f \quad \text{if } f \text{ is a primitive} \tag{12}$$

$$T[\![f_1 \circ f_2]\!] \Rightarrow T[\![f_1]\!] \circ T[\![f_2]\!] \quad \text{if no other rule applies} \tag{13}$$

These additional rules allow some facts about the particular primitives involved in a composition or conditional to be applied. (Rule 11 is a special case of simplifying a conditional using facts about its predicate—e.g., $true \rightarrow p; q \equiv p$.) Note that compositions may remain in the program if the rules are not strong enough to deal with composition of primitive functions (rule 13). In general this will be the case (e.g., consider the program $+ \circ s1$).

This example is adapted from Wadler's paper. Consider the following program to recursively reverse the branches of a tree:

def flip $\equiv$ ispair $\rightarrow$ [flip $\circ$ s2, flip $\circ$ s1]; id

1. $T[\![$flip $\circ$ flip$]\!]$
2. $T[\![$flip $\circ$ ispair $\rightarrow$ [flip $\circ$ s2, flip $\circ$ s1]; id$]\!]$ def
3. $T[\![$ispair $\rightarrow$ flip $\circ$ [flip $\circ$ s2, flip $\circ$ s1]; flip $\circ$ id$]\!]$ (1)
4. $T[\![$ispair$]\!] \rightarrow T[\![$flip $\circ$ [flip $\circ$ s2, flip $\circ$ s1]$]\!]; T[\![$id $\circ$ flip$]\!]$ (5)
5. ispair $\rightarrow T[\![$flip $\circ$ [flip $\circ$ s2, flip $\circ$ s1]$]\!]; T[\![$id $\circ$ flip$]\!]$ (12)
6. ispair $\rightarrow T[\![$flip $\circ$ [flip $\circ$ s2, flip $\circ$ s1]$]\!]; T[\![$flip$]\!]$ (8)
7. ispair $\rightarrow T[\![$flip $\circ$ [flip $\circ$ s2, flip $\circ$ s1]$]\!]; T[\![$ispair $\rightarrow$ [flip $\circ$ s2, flip $\circ$ s1]; id$]\!]$ def
8. ispair $\rightarrow T[\![$flip $\circ$ [flip $\circ$ s2, flip $\circ$ s1]$]\!]$; id (11?)
9. ispair $\rightarrow T[\![$(ispair $\rightarrow$ [flip $\circ$ s2, flip $\circ$ s1]; id) $\circ$ [flip $\circ$ s2, flip $\circ$ s1]$]\!]$; id def
10. ispair $\rightarrow T[\![$ispair $\rightarrow$ [flip $\circ$ s2, flip $\circ$ s1] $\circ$ [flip $\circ$ s2, flip $\circ$ s1]; id $\circ$ [flip $\circ$ s2, flip $\circ$ s1]$]\!]$; id (3)
11. ispair $\rightarrow T[\![[$flip $\circ$ s2, flip $\circ$ s1] $\circ$ [flip $\circ$ s2, flip $\circ$ s1]$]\!]$; id (11?)
12. ispair $\rightarrow T[\![[$flip $\circ$ s2 $\circ$ [flip $\circ$ s2, flip $\circ$ s1], flip $\circ$ s1 $\circ$ [flip $\circ$ s2, flip $\circ$ s1]$]]\!]$; id (1)
13. ispair $\rightarrow [T[\![$flip $\circ\, T[\![$s2 $\circ$ [flip $\circ$ s2, flip $\circ$ s1]$]\!]], T[\![$flip $\circ\, T[\![$s1 $\circ$ [flip $\circ$ s2, flip $\circ$ s1]$]\!]]\!]]$; id (6,6)
14. ispair $\rightarrow [T[\![$flip $\circ$ flip $\circ$ s1$]\!], T[\![$flip $\circ$ flip $\circ$ s2$]\!]]$; id (10,10)
15. def $h \equiv$ ispair $\rightarrow [h \circ$ s1, $h \circ$ s2]; id

The last line follows because flip $\circ$ flip is a term that has been encountered before in the derivation; in fact, it is the original term. Thus, the original function is redefined in terms of itself, but without the intermediate data structures.

Note that the precedence of rules is almost unnecessary; the rules are nearly unambiguous. Only rules 1 and 8 introduce ambiguity.

```
                ( assuming that f == DEL:isseq @ f )
                (no gamma assumptions are needed.)

( pp @ [ C:f@g , h ]   ==  f@[g,h]
                (no gamma assumptions are needed.)

intsto  ==  for:<Ind,id>

distr @ [ for:<E,h> , f ]   ==   for : < [E,f@Arg] , h >
                ( usual gamma assumptions)

distl  ==  for:< [ s1@Arg , Rel@[s2@Arg,Ind] ] , len@s2 >

AA:f @ for:<E,h>  ==  for:<f@E,h>
                ( usual gamma assumptions)

for:<E,h>@f  ==  for:<E@[f@Arg,Ind],h@f>
                (usual gamma assumptions.)
```

From "Some for derivations . . ."

```
trans

  (1)  ==  ( cons @ AA:(AA@C:sel) @ insto @ len @ s1 )   :  id

  (2)  ==  app @ [  cons @ AA:(AA@C:sel) @ insto @ len @ s1  ,  id ]

  (3)  ==  AA:app @ distr @ [ AA:(AA@C:sel) @ insto @ len @ s1 , id ]

  (4)  ==  AA:app @ distr @ [ AA:(AA@C:sel) @ for:<Ind,id> @ len @ s1 , id ]

  (5)  ==  AA:app @ distr @
               [ AA:(AA@C:sel) @ for:<Ind@[len@s1@Arg,Ind],len@s1> , id ]
```

```
  (6)  ==  AA:app @ distr @ [ AA:(AA@C:sel) @ for:<Ind,len@s1> , id ]

  (7)  ==  AA:app @ distr @ [ for:< AA@C:sel@Ind , len@s1 > , id ]

  (8)  ==  AA:app @ for:< [ AA@C:sel@Ind , id@Arg ] , len@s1 >

  (9)  ==  for:< app @ [ AA@C:sel@Ind , Arg ] , len@s1 >

 (10)  ==  for:< AA:app @ distl @ [ C:sel@Ind , Arg ] , len@s1 >

 (11)  ==  for:< AA:app @
           for:<[s1@Arg,Rel@[s2@Arg,Ind]],len@s2> @ [C:sel@Ind,Arg] , len@s1 >

 (12)  ==  for:< AA:app @
           for:<[s1@Arg,Rel@[s2@Arg,Ind]]@[[C:sel@Ind,Arg]@Arg,Ind],
                      len@s2@[C:sel@Ind,Arg]> , len@s1 >

 (13)  ==  for:< AA:app @
           for:< [s1@Arg,Rel@[s2@Arg,Ind]] @ [[C:sel@Ind_2,Arg_2],Ind_1] ,
                      len@Arg_1 > , len@s1 >

 (14)  ==  for:< AA:app@for:<[C:sel@Ind_2,Rel@[Arg_2,Ind_1]],len@Arg_1>,len@s1>

 (15)  ==  for:< for:<app@[C:sel@Ind_2,Rel@[Arg_2,Ind_1]],len@Arg_1 >, len@s1 >

 (16)  ==  for:< for : < sel @ [Ind_2,Rel@[Arg_2,Ind_1]],len@Arg_1 > , len@s1 >

 (17)  ==  for:< for : < Rel @ [Rel@[Arg_2,Ind_1],Ind_2],len@Arg_1 > , len@s1 >

 (18)  ==  for:< for : < /:Rel @ [ Arg_2,Ind_1,Ind_2 ] , len@Arg_1 > , len@s1 >
```