Shortcut Fusion and Tuples

Josef Svenningsson

Talk Overview

- Crash course on Shortcut Fusion
- Wintermeeting talk:
  Fusion with functions returning multiple lists in tuples
- Today:
  Removing tuples from recursive functions, Shortcut fusion style

Shortcut Fusion Crash Course

- A common functional programming idiom: composing highly reuseable components
- This incurs an overhead, the intermediate data structure
- Shortcut Fusion aim to remove this overhead
- Classical (toy) example

```
sum (map square [1..n])
```

The Essence

```
foldr c n (build g) = g c n
```

foldr

- `foldr` is a common function in functional programming
- It is a very powerful function for consuming lists. It can be used to define many list processing functions

```
foldr :: (a → b → b) → b → [a] → b
```

build

- `build` is a special function used only in shortcut fusion.
- `build` is used for constructing lists
- Given a function which takes to arguments and constructs something with them, `build` will give that function (:) and [] so that it produces lists

```
build :: (forall b . (a → b → b) → b → b) → [a]
build g = g (:) []
```

The rule again

- The rule removes the list produced by `build` and consumed by `foldr`

```
foldr c n (build g) = g c n
```

Making the rule useful

- But the rule looks utterly useless! It only involves two functions. Do we have to define new rules for all pairs of functions?

- Idea: Define functions in terms of `foldr` and `build`. Then inline the function definitions
- `foldr` and `build` are particularly good in this respect because many functions can be defined in terms of them

Defining list functions

- As many list processing function as possible should be defined in terms of `foldr` and `build`

```
map :: (a → b) → [a] → [b]
map f xs = build (\c n → foldr (c . f) n xs)

filter :: (a → Bool) → [a] → [a]
filter p xs = build (\c n →
  foldr (\a b → if p a
                then c a b
                else b) n xs)

length :: [a] → Int
length xs = foldr (\_ ln → 1 + ln) 0 xs
```

Good Producers and Consumers

- We call functions which produce lists using `build` **Good Producers**.
- We call functions which consume lists using `foldr` **Good Consumers**.
- Example: `map` is both a good producer and a good consumer

```
map f xs = build (\c n → foldr (c . f) n xs)
```

- `words` is a good producer and `length` is a good consumer

Good Producers and Consumers

- Whenever a Good Consumer is applied to a Good Consumer the intermediate list is removed

- The reason is that when the definitions of the functions are inlined then `foldr` will be applied to `build` and the `foldr/build` rule can fire

Example Fusion

```
map f (map g xs)
```

Example Fusion

```
map f (map g xs)
=
build (\c1 n1 →
  foldr (c1 . f) n1 (build
    (\c2 n2 → foldr (c2 . g) n2 xs)))
```

Example Fusion

```
map f (map g xs)
=
build (\c1 n1 →
  foldr (c1 . f) n1 (build
    (\c2 n2 → foldr (c2 . g) n2 xs)))
```

Example Fusion

```
map f (map g xs)
=
build (\c1 n1 →
  foldr (c1 . f) n1 (build
    (\c2 n2 → foldr (c2 . g) n2 xs)))
=
build (\c1 n1 →
  (\c2 n2 → foldr (c2 . g) n2 xs) (c1 . f) n1)
```

Example Fusion

```
map f (map g xs)
=
build (\c1 n1 →
  foldr (c1 . f) n1 (build
    (\c2 n2 → foldr (c2 . g) n2 xs)))
=
build (\c1 n1 →
  (\c2 n2 → foldr (c2 . g) n2 xs) (c1 . f) n1)
=
build (\c1 n2 → foldr (c1 . f . g) n1 xs)
```

Example Fusion

```
map f (map g xs)
=
build (\c1 n1 →
  foldr (c1 . f) n1 (build
    (\c2 n2 → foldr (c2 . g) n2 xs)))
=
build (\c1 n1 →
  (\c2 n2 → foldr (c2 . g) n2 xs) (c1 . f) n1)
=
build (\c1 n2 → foldr (c1 . f . g) n1 xs)
```

The same as `map (f . g) xs`

Implementing Fusion

- GHC, the standard Haskell compiler provides a way for the programmer to add new optimizations to the compiler
- This is specified using rewrite rules

```
{-# RULES
  "foldr/build"
  forall c n (g :: forall l. (a -> l -> l) -> l -> l) .
  foldr c n (build g) = g c n
 #-}
```

End of Crash Course

- Any questions so far?

Main Subject

Removing tuples from recursive functions,
Shortcut fusion style

Recursion and Tuples

- Recursive functions returning tuples are difficult to make efficient
- There are several papers on how to improve such programs

```
partition :: (a → Bool) → [a] → ([a],[a])
partition p [] = ([],[])
partition p (a:as) =
  let (bs,cs) = partition p as
  in if p a
       then (a:bs,cs)
       else (bs,a:cs)
```

Removing tuples

- One way to deal with these tuples is to remove them completely whenever possible
- In this example we don't actually need to compute the whole tuple

```
snd (partition p ls)
```

- This situation doesn't come up in programmers' code but might show up during optimizations in the compiler

Build for Tuples

- How should we formulate build for tuples?
- Here's a first stab

```
buildP :: (forall p . a → b → p) → (a,b)
buildP g = g (,)
```

- The idea is to try to transfer the intuition from the list case where we pass the constructors of the data type

Failed first attempt

- Our first attempt fails because in the recursive call we need to take apart the tuple

```
partition :: (a → Bool) → [a] → ([a],[a])
partition p [] = ([],[])
partition p (a:as) =
  let (bs,cs) = partition p as
  in if p a
       then (a:bs,cs)
       else (bs,a:cs)
```

Second attempt

- Ok, so our build function must provide a way to deconstruct tuples

```
buildP :: (forall p . (a → b → p) →
                      (p → a) →
                      (p → b))
          → (a,b)
buildP g = g (,) fst snd
```

Fusion rule

- So, what should our fusion rule look like now?

```
{-# RULES
  "buildP/fst"
  forall (g :: forall p . (a → b → p) →
                          (p → a) →
                          (p → b) →
                          p) .
  fst (buildP g) = g const id (error ”snd”)
 #-}
```

- If we want to remove the second part of the pair, there is no way to project it out. We must return error.

Failed second attempt

- However, this rules is not correct in the presence of seq.
- In order to make this rule correct we would have to make severe restrictions on how it can be used.
- Proving it correct with Free Theorems is out of the question
- Can we do better?

Key Insight

- Our second attempt was not that far off
- The key insight about the functions we are trying to transform is this:

Whenever we take the pair apart we put it back together immediately

Third Attempt

- Instead of taking the pair apart completely we can simply provide a way to change the components of the pair

```
buildP ::
 (forall p .
   (a → b → p)
 → ((a → a) → (b → b) → p → p)
 → p)
       → (a,b)
buildP g = g (,) (***)
```

Fusion rule

- Here are some fusion rules

```
fst (buildP g) = g const (\mapA _ a -> mapA a)

swap (buildP g)
  = g (\a b -> (b,a))
      (\mapA mapB (b,a) -> (mapB b,mapA a))
```

Fusable Functions

```
partP pair mapP p []     = pair [] []
partP pair mapP p (a:as)
  | p a  = mapP (a:) id (partPM pair mapP p as)
  | True = mapP id (a:) (partPM pair mapP p as)

partition p ls
  = buildP (\pair mapP → partP pair mapP p ls)
-- ---------------------------------
filterPM p ls = fst (partition p ls)
-- ---------------------------------
uz pair mapP [] = pair [] []
uz pair mapP ((a,b):ls)
  = mapP (a:) (b:) (uz pair mapP ls)

unzip ls = buildP (\pair mapP → uz pair mapP ls)
```

Coexisting with list fusion

- The functions I have shown operates on lists
- Can we apply list fusion at the same time as tuple fusion?
- YES!
- List fusion and tuple fusion are orthorgonal