Rebooting Supercompilation for Haskell

Ömer S. Ağacan
oagacan@indiana.edu

Ryan R. Newton
rrnewton@indiana.edu

August 21, 2015

Rebooting Supercompilation for Haskell

- An overview of supercompilation and problems associated with it.

Rebooting Supercompilation for Haskell

- An overview of supercompilation and problems associated with it.
- Why it's worth rebooting, and why GHC is a great compiler to base this work on.

Rebooting Supercompilation for Haskell

- An overview of supercompilation and problems associated with it.
- Why it's worth rebooting, and why GHC is a great compiler to base this work on.
- My preliminary work, and problems I encountered while working on a GHC plugin.

Rebooting Supercompilation for Haskell

- An overview of supercompilation and problems associated with it.
- Why it's worth rebooting, and why GHC is a great compiler to base this work on.
- My preliminary work, and problems I encountered while working on a GHC plugin.
- What's next? My research goals.

Supercompilation: An overview

- The paper that describes the idea in English: "The Concept of a Supercompiler" Turchin [1986].

Supercompilation: An overview

- The paper that describes the idea in English: "The Concept of a Supercompiler" Turchin [1986].
- High-level idea:

Supercompilation: An overview

- The paper that describes the idea in English: "The Concept of a Supercompiler" Turchin [1986].
- High-level idea:
  - Evaluate programs in compile-time, while making the most out of known inputs and definitions.

Supercompilation: An overview

- The paper that describes the idea in English: "The Concept of a Supercompiler" Turchin [1986].
- High-level idea:
  - Evaluate programs in compile-time, while making the most out of known inputs and definitions.
    - Definitions of used functions.

Supercompilation: An overview

- The paper that describes the idea in English: "The Concept of a Supercompiler" Turchin [1986].
- High-level idea:
    - Evaluate programs in compile-time, while making the most out of known inputs and definitions.
        - Definitions of used functions.
        - Statically known arguments of functions.

Supercompilation: An overview

- The paper that describes the idea in English: "The Concept of a Supercompiler" Turchin [1986].
- High-level idea:
  - Evaluate programs in compile-time, while making the most out of known inputs and definitions.
    - Definitions of used functions.
    - Statically known arguments of functions.
    - When branching, propagate learned information through branches and make use of that information while compiling branches.

Supercompilation: An overview

- The paper that describes the idea in English: "The Concept of a Supercompiler" Turchin [1986].
- High-level idea: (contd)
  - Evaluate programs in compile-time, while making the most out of known inputs and definitions.
    - Most of the time the goal is to generate more efficient programs.
      (but see Klyuchnikov and Romanenko [2010] for a different use of supercompilation)

Supercompilation: An overview

- One nice idea here is to base supercompilation algorithm on the language's operational semantics, as done in Bolingbroke and Peyton Jones [2010].

Supercompilation: An overview

- One nice idea here is to base supercompilation algorithm on the language's operational semantics, as done in Bolingbroke and Peyton Jones [2010].
- This optimizes in the sense that:
  If we have a programs $\mathcal{P}_1$ and $\mathcal{P}_2$, and
  $\mathcal{P}_1 \Downarrow v$ in N steps and
  $\mathcal{P}_2 \Downarrow v$ in M steps,
  we consider $\mathcal{P}_2$ optimized if $M < N$.

Supercompilation: An overview

- One nice idea here is to base supercompilation algorithm on the language's operational semantics, as done in Bolingbroke and Peyton Jones [2010].
- This optimizes in the sense that:
  If we have a programs $\mathcal{P}_1$ and $\mathcal{P}_2$, and
  $\mathcal{P}_1 \Downarrow v$ in N steps and
  $\mathcal{P}_2 \Downarrow v$ in M steps,
  we consider $\mathcal{P}_2$ optimized if $M < N$.
- An approximation. It's very unlikely that all of the rules have same costs.

An example

```haskell
mapOfMap f g = (.) (map f) (map g)
```

An example

```
mapOfMap f g = (.) (map f) (map g)
```

We compile RHS of this definition, and we introduce a new definition at each step. We give it a fresh name, and add arguments for free variables in the expression.

An example

```
mapOfMap f g = (.) (map f) (map g)
```

We compile RHS of this definition, and we introduce a new definition at each step. We give it a fresh name, and add arguments for free variables in the expression.

When we get stuck, we keep evaluating sub-expressions.

An example

```
mapOfMap f g = (.) (map f) (map g)
```

An example

```
mapOfMap f g = (.) (map f) (map g)
```

Lookup `(.)`: β-reduction:

An example

```
mapOfMap f g = (.) (map f) (map g)
```

Lookup (.): β-reduction:

```
h1 f g = (\f1 -> \f2 -> \a -> f1 (f2 a))
              (map f) (map g)
```

An example

```
mapOfMap f g = (.) (map f) (map g)
```

Lookup `(.)`: β-reduction:

```
h1 f g = (\f1 -> \f2 -> \a -> f1 (f2 a))
              (map f) (map g)
```

β-reduction:

An example

```
mapOfMap f g = (.) (map f) (map g)
```

Lookup (.): β-reduction:

```
h1 f g = (\f1 -> \f2 -> \a -> f1 (f2 a))
              (map f) (map g)
```

β-reduction:

```
h2 f g = (\f2 a -> (map f) (f2 a)) (map g)
```

An example

```
mapOfMap f g = (.) (map f) (map g)
```

Lookup `(.)`: β-reduction:

```
h1 f g = (\f1 -> \f2 -> \a -> f1 (f2 a))
             (map f) (map g)
```

β-reduction:

```
h2 f g = (\f2 a -> (map f) (f2 a)) (map g)
```

β-reduction:

An example

`mapOfMap f g = (.) (map f) (map g)`

Lookup `(.)`: β-reduction:

```
h1 f g = (\f1 -> \f2 -> \a -> f1 (f2 a))
            (map f) (map g)
```

β-reduction:

`h2 f g = (\f2 a -> (map f) (f2 a)) (map g)`

β-reduction:

`h3 f g a = (map f) (map g a)`

```haskell
h3 f g a = (map f) (map g a)
```

```
h3 f g a = (map f) (map g a)
```

Lookup `map`:

```
h3 f g a = (map f) (map g a)
```

Lookup `map`:

```
h5 f g a = ((\f -> \l ->
               case l of
                 []    -> []
                 h : t -> f h : map f t) f) (map g a)
```

```
h3 f g a = (map f) (map g a)
```

Lookup `map`:

```
h5 f g a = ((\f -> \l ->
              case l of
                []    -> []
                h : t -> f h : map f t) f) (map g a)
```

β-reduction (twice):

```haskell
h3 f g a = (map f) (map g a)
```

Lookup `map`:

```haskell
h5 f g a = ((\f -> \l ->
                case l of
                  []     -> []
                  h : t -> f h : map f t) f) (map g a)
```

β-reduction (twice):

```haskell
h6 f g a = case (map g a) of
             []     -> []
             h : t -> f h : map f t
```

```
h3 f g a = (map f) (map g a)
```

Lookup `map`:

```
h5 f g a = ((\f -> \l ->
                case l of
                  []     -> []
                  h : t -> f h : map f t) f) (map g a)
```

β-reduction (twice):

```
h6 f g a = case (map g a) of
             []     -> []
             h : t -> f h : map f t
```

Lookup `map`, beta reduction:

```
h3 f g a = (map f) (map g a)
```

Lookup `map`:

```
h5 f g a = ((\f -> \l ->
                case l of
                  []    -> []
                  h : t -> f h : map f t) f) (map g a)
```

β-reduction (twice):

```
h6 f g a = case (map g a) of
             []    -> []
             h : t -> f h : map f t
```

Lookup `map`, beta reduction:

```
h7 f g a = case (case a of
                   []      -> []
                   h1 : t1 -> g h1 : map g t1) of
             []      -> []
             h0 : t0 -> f h0 : map f t0
```

Case-of-case transformation: (Jones and Santos [1998])

Case-of-case transformation: (Jones and Santos [1998])

```
h8 f g a = case a of
             [] -> case [] of
                     []      -> []
                     h0 : t0 -> f h0 : map f t0
             h1 : t1 -> case (g h1 : map g t1) of
                          []      -> []
                          h0 : t0 -> f h0 : map f t0
```

Case-of-case transformation: (Jones and Santos [1998])

```
h8 f g a = case a of
             [] -> case [] of
                     []      -> []
                     h0 : t0 -> f h0 : map f t0
             h1 : t1 -> case (g h1 : map g t1) of
                          []      -> []
                          h0 : t0 -> f h0 : map f t0
```

At this point we consider all branches, let's start with first one:

Case-of-case transformation: (Jones and Santos [1998])

```
h8 f g a = case a of
             [] -> case [] of
                     []      -> []
                    h0 : t0 -> f h0 : map f t0
            h1 : t1 -> case (g h1 : map g t1) of
                         []      -> []
                         h0 : t0 -> f h0 : map f t0
```

At this point we consider all branches, let's start with first one:

```
            case [] of
              [] -> []
              h0 : t0 -> f h0 : map f t0
```

Case-of-case transformation: (Jones and Santos [1998])

```
h8 f g a = case a of
             [] -> case [] of
                        []      -> []
                        h0 : t0 -> f h0 : map f t0
             h1 : t1 -> case (g h1 : map g t1) of
                             []      -> []
                             h0 : t0 -> f h0 : map f t0
```

At this point we consider all branches, let's start with first one:

```
case [] of
  [] -> []
  h0 : t0 -> f h0 : map f t0
```

Known case reduction evaluates this to it's final form, and we update our expression to:

```
h9 f g a = case a of
             []      -> []
             h1 : t1 -> case (g h1 : map g t1) of
                          []      -> []
                          h0 : t0 -> f h0 : map f t0
```

```
h9 f g a = case a of
             []      -> []
             h1 : t1 -> case (g h1 : map g t1) of
                          []      -> []
                          h0 : t0 -> f h0 : map f t0
```

Second branch:

```
h9 f g a = case a of
             []      -> []
             h1 : t1 -> case (g h1 : map g t1) of
                          []      -> []
                          h0 : t0 -> f h0 : map f t0
```

Second branch:

```
case (g h1 : map g t1) of
  []      -> []
  h0 : t0 -> f h0 : map f t0
```

```
h9 f g a = case a of
            []      -> []
            h1 : t1 -> case (g h1 : map g t1) of
                         []      -> []
                         h0 : t0 -> f h0 : map f t0
```

Second branch:

```
case (g h1 : map g t1) of
  []      -> []
  h0 : t0 -> f h0 : map f t0
```

Since `h0` and `t0` are linear in the RHS of the second branch, we can just do substitution, without introducing lets:

```
h9 f g a = case a of
             []       -> []
             h1 : t1 -> case (g h1 : map g t1) of
                          []       -> []
                          h0 : t0 -> f h0 : map f t0
```

Second branch:

```
case (g h1 : map g t1) of
  []       -> []
  h0 : t0 -> f h0 : map f t0
```

Since `h0` and `t0` are linear in the RHS of the second branch, we can just do substitution, without introducing lets:

```
f (g h1) : map f (map g t1)
```

```
h9 f g a = case a of
             []       -> []
             h1 : t1 -> case (g h1 : map g t1) of
                          []       -> []
                          h0 : t0 -> f h0 : map f t0
```

Second branch:

```
case (g h1 : map g t1) of
  []       -> []
  h0 : t0 -> f h0 : map f t0
```

Since `h0` and `t0` are linear in the RHS of the second branch, we can just do substitution, without introducing lets:

```
f (g h1) : map f (map g t1)
```

So we now have:

```
h9 f g a = case a of
             []      -> []
             h1 : t1 -> case (g h1 : map g t1) of
                          []      -> []
                          h0 : t0 -> f h0 : map f t0
```

Second branch:

```
case (g h1 : map g t1) of
  []      -> []
  h0 : t0 -> f h0 : map f t0
```

Since `h0` and `t0` are linear in the RHS of the second branch, we can just do substitution, without introducing lets:

```
f (g h1) : map f (map g t1)
```

So we now have:

```
h10 f g a = case a of
              []      -> []
              h1 : t1 -> f (g h1) : map f (map g t1)
```

As in `h8`, we again consider alternatives. There's nothing to do in first branch.

As in `h8`, we again consider alternatives. There's nothing to do in first branch.

```
f (g h1) : map f (map g t1)
```

As in `h8`, we again consider alternatives. There's nothing to do in first branch.

```
f (g h1) : map f (map g t1)
```

This term is already in WHFN, but we consider sub-terms.

As in `h8`, we again consider alternatives. There's nothing to do in first branch.

```
f (g h1) : map f (map g t1)
```

This term is already in WHFN, but we consider sub-terms.

```
f (g h1)
```

As in `h8`, we again consider alternatives. There's nothing to do in first branch.

```
f (g h1) : map f (map g t1)
```

This term is already in WHFN, but we consider sub-terms.

```
f (g h1)
```

We can't do anything about this, all names are free. We consider the second sub-term.

```
map f (map g t1)
```

```
map f (map g t1)
```

This looks a lot like one of the terms we compiled before. Remember `h3`:

```
map f (map g t1)
```

This looks a lot like one of the terms we compiled before. Remember `h3`:

```
h3 f g a = map f (map g a)
```

```
map f (map g t1)
```

This looks a lot like one of the terms we compiled before. Remember `h3`:

```
h3 f g a = map f (map g a)
```

Our current term is just a renaming of `h3`. We started with `h3` and came across the same term. If we continue we'll loop.

```
map f (map g t1)
```

This looks a lot like one of the terms we compiled before. Remember `h3`:

```
h3 f g a = map f (map g a)
```

Our current term is just a renaming of `h3`. We started with `h3` and came across the same term. If we continue we'll loop.

Instead, we generate a call to `h3`.

```
map f (map g t1)
```

This looks a lot like one of the terms we compiled before. Remember `h3`:

```
h3 f g a = map f (map g a)
```

Our current term is just a renaming of `h3`. We started with `h3` and came across the same term. If we continue we'll loop.

Instead, we generate a call to `h3`.

```
h3 f g t1
```

```
map f (map g t1)
```

This looks a lot like one of the terms we compiled before. Remember `h3`:

```
h3 f g a = map f (map g a)
```

Our current term is just a renaming of `h3`. We started with `h3` and came across the same term. If we continue we'll loop.

Instead, we generate a call to `h3`.

```
h3 f g t1
```

Since that `h11` is optimized version of `h3`, we replace the call to `h3` with a call to `h11`, and we have our final definition:

```
map f (map g t1)
```

This looks a lot like one of the terms we compiled before. Remember `h3`:

```
h3 f g a = map f (map g a)
```

Our current term is just a renaming of `h3`. We started with `h3` and came across the same term. If we continue we'll loop.

Instead, we generate a call to `h3`.

```
h3 f g t1
```

Since that `h11` is optimized version of `h3`, we replace the call to `h3` with a call to `h11`, and we have our final definition:

```
h11 f g a = case a of
              []      -> []
              h1 : t1 -> f (g h1) : h11 f g t1
```

Short digression

This looked a lot similar to deforestation(Wadler [1988]).

Short digression

This looked a lot similar to deforestation(Wadler [1988]).

Indeed, all of the steps we took here are described in the original deforestation paper.

Short digression

This looked a lot similar to deforestation(Wadler [1988]).

Indeed, all of the steps we took here are described in the original deforestation paper.

Some of the important differences are:

Short digression

This looked a lot similar to deforestation(Wadler [1988]).

Indeed, all of the steps we took here are described in the original deforestation paper.

Some of the important differences are:

- No linearity restriction. (in the linear case they would do similar things – see Sørensen et al. [1994] for a comparison)

Short digression

This looked a lot similar to deforestation(Wadler [1988]).

Indeed, all of the steps we took here are described in the original deforestation paper.

Some of the important differences are:

- No linearity restriction. (in the linear case they would do similar things – see Sørensen et al. [1994] for a comparison)
- We do generalization. (not demonstrated here)

Operations of a supercompiler - Driving

We evaluated the program, and while doing that we were careful with previously evaluated terms: Driving.

Operations of a supercompiler - Splitting

When we're stuck because the expression we do pattern matching on couldn't take any more steps, we evaluated branches:

Operations of a supercompiler - Splitting

When we're stuck because the expression we do pattern matching on couldn't take any more steps, we evaluated branches:

```
case a of
  []      -> []
  h1 : t1 -> f (g h1) : map f (map g t1)
```

Operations of a supercompiler - Splitting

When we're stuck because the expression we do pattern matching on couldn't take any more steps, we evaluated branches:

```
case a of
  []       -> []
  h1 : t1 -> f (g h1) : map f (map g t1)
```

When we're stuck because the term is in WHNF, we evaluated subterms:

Operations of a supercompiler - Splitting

When we're stuck because the expression we do pattern matching on couldn't take any more steps, we evaluated branches:

```
case a of
  []      -> []
  h1 : t1 -> f (g h1) : map f (map g t1)
```

When we're stuck because the term is in WHNF, we evaluated subterms:

```
f (g h1) : map f (map g t1)
```

Operations of a supercompiler - Splitting

When we're stuck because the expression we do pattern matching on couldn't take any more steps, we evaluated branches:

```
case a of
  []      -> []
  h1 : t1 -> f (g h1) : map f (map g t1)
```

When we're stuck because the term is in WHNF, we evaluated subterms:

```
f (g h1) : map f (map g t1)
```

This is called splitting.

Operations of a supercompiler - matching

We need to "compare" current expression with our history of expressions, to prevent loops, and generate optimized functions.

Operations of a supercompiler - matching

We need to "compare" current expression with our history of expressions, to prevent loops, and generate optimized functions.

```
h3 f g a = map f (map g a)
...
h10 f g a = ... map f (map g t1) ...
```

Operations of a supercompiler - matching

We need to "compare" current expression with our history of expressions, to prevent loops, and generate optimized functions.

```
h3 f g a = map f (map g a)
...
h10 f g a = ... map f (map g t1) ...
```

We also need a way to call optimized function when this happens.

Operations of a supercompiler - matching

We need to "compare" current expression with our history of expressions, to prevent loops, and generate optimized functions.

```
h3 f g a = map f (map g a)
...
h10 f g a = ... map f (map g t1) ...
```

We also need a way to call optimized function when this happens.

Matching, returns all the necessary information to replace current expression with a function call to optimized function.

Operations of a supercompiler - termination checking

Something we haven't demonstrated so far.

Operations of a supercompiler - termination checking

Something we haven't demonstrated so far.

```
reverse_acc []      acc = acc
reverse_acc (h : t) acc = reverse_acc t (h : acc)
```

Operations of a supercompiler - termination checking

Something we haven't demonstrated so far.

```
reverse_acc []      acc = acc
reverse_acc (h : t) acc = reverse_acc t (h : acc)

h0 lst = reverse_acc (reverse_acc lst []) []
```

Operations of a supercompiler - termination checking

```
h0 lst = reverse_acc (reverse_acc lst []) []
...
h5 lst =
  case lst of
    []      -> []
    h1 : t1 ->
      case (reverse_acc t1 (h1 : [])) of
        []      -> []
        h0 : t0 -> reverse_acc t0 (h0 : [])
...
h_ lst = ... reverse_acc t1 (h1 : []) ...
...
h_ lst = ... reverse_acc t2 (h2 : h1 : []) ...
...
```

Operations of a supercompiler - termination checking

```
reverse_acc []      acc = acc
reverse_acc (h : t) acc = reverse_acc t (h : acc)
```

Operations of a supercompiler - termination checking

```
reverse_acc []      acc = acc
reverse_acc (h : t) acc = reverse_acc t (h : acc)
```

The growing argument(accumulator) makes this example tricky.

Operations of a supercompiler - termination checking

```
reverse_acc []      acc = acc
reverse_acc (h : t) acc = reverse_acc t (h : acc)
```

The growing argument(accumulator) makes this example tricky.

We need a termination checker to stop in cases like this.

Operations of a supercompiler - termination checking

```
reverse_acc []      acc = acc
reverse_acc (h : t) acc = reverse_acc t (h : acc)
```

The growing argument(accumulator) makes this example tricky.

We need a termination checker to stop in cases like this.

What to do after stopping like this is another story.
(see Bolingbroke [2013])

Operations of a supercompiler - issues

Each one has tricky problems.

Operations of a supercompiler - issues

Each one has tricky problems.

Splitter:

Operations of a supercompiler - issues

Each one has tricky problems.

Splitter:

- We want to propagate information to sub-terms.

Operations of a supercompiler - issues

Each one has tricky problems.

Splitter:

- We want to propagate information to sub-terms.
- Propagate too much: We end up duplicating work.

Operations of a supercompiler - issues

Each one has tricky problems.

Splitter:

- We want to propagate information to sub-terms.
- Propagate too much: We end up duplicating work.
- Example: (from Bolingbroke [2013])

```
let n = fib 100
    b = n + 1
    c = n + 2
 in (b, c)
```

Operations of a supercompiler - issues

Each one has tricky problems.

Splitter:

- We want to propagate information to sub-terms.
- Propagate too little: We miss optimization opportunities.

Operations of a supercompiler - issues

Each one has tricky problems.

Splitter:

- We want to propagate information to sub-terms.
- Propagate too little: We miss optimization opportunities.
- Example: (from Bolingbroke [2013])

```
let map = ...
    ys = map f zs
    xs = map g ys
 in Just xs
```

Operations of a supercompiler - issues

Each one has tricky problems.

Operations of a supercompiler - issues

Each one has tricky problems.

Matcher:

Operations of a supercompiler - issues

Each one has tricky problems.

Matcher:

- Syntactic equality(modulo α-renaming) almost never holds.

Operations of a supercompiler - issues

Each one has tricky problems.

Matcher:

- Syntactic equality(modulo α-renaming) almost never holds.
- We may end up sharing work, which is not a good thing in general(increases residency, see Chitil [1997]) (example from Bolingbroke [2013])

Operations of a supercompiler - issues

Each one has tricky problems.

Matcher:

- Syntactic equality(modulo α-renaming) almost never holds.
- We may end up sharing work, which is not a good thing in general(increases residency, see Chitil [1997]) (example from Bolingbroke [2013])
- ```
  S0 =
      let a = fib y
          b = fib y
       in (a, b)

  S1 = let a = fib y in (a, a)
  ```
  ```

Operations of a supercompiler - issues

- "We do not attempt to supercompile the full Nofib suite because the other Nofib benchmarks are considerably more complicated and generally suffer from extremely long supercompilation times."

Operations of a supercompiler - issues

- "We do not attempt to supercompile the full Nofib suite because the other Nofib benchmarks are considerably more complicated and generally suffer from extremely long supercompilation times."
- Jonsson [201?] focuses on compilation performance, and reports <*3 seconds* for all the small programs from nofib.

Latest work

- Bolingbroke [2013] laid out a great framework, with nicely defined components. (Splitter, matcher, termination checker etc.)

Latest work

- Bolingbroke [2013] laid out a great framework, with nicely defined components. (Splitter, matcher, termination checker etc.)
- Problems and current solutions are documented nicely.

Latest work

- Bolingbroke [2013] laid out a great framework, with nicely defined components. (Splitter, matcher, termination checker etc.)
- Problems and current solutions are documented nicely.
- We don't have any solutions that work well on *all* programs.

Latest work

- Bolingbroke [2013] laid out a great framework, with nicely defined components. (Splitter, matcher, termination checker etc.)
- Problems and current solutions are documented nicely.
- We don't have any solutions that work well on *all* programs.
- We don't have a usable implementation.

It's worth rebooting!

- Bolingbroke [2013] shows some great potential.

It's worth rebooting!

- Bolingbroke [2013] shows some great potential.
- Up to -95.1% runtime improvement.

It's worth rebooting!

- Bolingbroke [2013] shows some great potential.
- Up to -95.1% runtime improvement.
- Up to -100.0% allocation improvement.

GHC Plugin API

- GHC plugin API can help here, we can implement a supercompiler as a plugin, get immediate feedback from users.

GHC Plugin API

- GHC plugin API can help here, we can implement a supercompiler as a plugin, get immediate feedback from users.
- No need to merge anything to GHC(in theory).

GHC Plugin API

- GHC plugin API can help here, we can implement a supercompiler as a plugin, get immediate feedback from users.
- No need to merge anything to GHC(in theory).
- But... GHC API feels like *exposed internals* rather than an *API*.

Problems with GHC Plugin API

- No easy ways to do most basic stuff: Moving terms around(substitutions), known-case reduction, case-of-case, etc. (all done in some parts of Core-to-Core passes, need to reverse engineer)

Problems with GHC Plugin API

- No easy ways to do most basic stuff: Moving terms around(substitutions), known-case reduction, case-of-case, etc. (all done in some parts of Core-to-Core passes, need to reverse engineer)
- No easy way to annotate Core syntax. Duplicating the syntax means duplicating huge amounts of code.

Problems with GHC Plugin API

- No easy ways to do most basic stuff: Moving terms around(substitutions), known-case reduction, case-of-case, etc. (all done in some parts of Core-to-Core passes, need to reverse engineer)
- No easy way to annotate Core syntax. Duplicating the syntax means duplicating huge amounts of code.
- Working on Core hard: Invariants are encoded as partial functions without any helpful error messages – if we're lucky, there's a NOTE.

Problems with GHC Plugin API

- No easy ways to do most basic stuff: Moving terms around(substitutions), known-case reduction, case-of-case, etc. (all done in some parts of Core-to-Core passes, need to reverse engineer)
- No easy way to annotate Core syntax. Duplicating the syntax means duplicating huge amounts of code.
- Working on Core hard: Invariants are encoded as partial functions without any helpful error messages – if we're lucky, there's a NOTE.
- Some things are not clear. (Types are first-class, but can I use them wherever I want? The definition allows this)

A more fundamental problem

Supercompilation is inherently a whole-program optimization technique. ("-O99")

A more fundamental problem

Supercompilation is inherently a whole-program optimization technique. ("-O99")

It can optimize just a single definition, but more definitions mean more optimizations.

A more fundamental problem

Supercompilation is inherently a whole-program optimization technique. ("-O99")

It can optimize just a single definition, but more definitions mean more optimizations.

GHC only saves definitions of small definitions, or definitions with `INLINE` or `INLINABLE`.

A more fundamental problem

Supercompilation is inherently a whole-program optimization technique. ("-O99")

It can optimize just a single definition, but more definitions mean more optimizations.

GHC only saves definitions of small definitions, or definitions with `INLINE` or `INLINABLE`.

Ideally we need all the definitions in a module in it's `.hi` file.

A more fundamental problem

Supercompilation is inherently a whole-program optimization technique. ("-O99")

It can optimize just a single definition, but more definitions mean more optimizations.

GHC only saves definitions of small definitions, or definitions with `INLINE` or `INLINABLE`.

Ideally we need all the definitions in a module in it's `.hi` file.

Idea: Use `-fexpose-all-unfoldings` all the time, distribute `base` with this option.

Roadmap

- Working on implementing the supercompiler described in Bolingbroke [2013].

Roadmap

- Working on implementing the supercompiler described in Bolingbroke [2013].
- (Hopefully) Improving GHC plugin API on the way.

Roadmap

- Working on implementing the supercompiler described in Bolingbroke [2013].
- (Hopefully) Improving GHC plugin API on the way.
- Collecting benchmark programs - send yours! (with expected optimizations)

Roadmap

- Working on implementing the supercompiler described in Bolingbroke [2013].
- (Hopefully) Improving GHC plugin API on the way.
- Collecting benchmark programs - send yours! (with expected optimizations)
- Once we have that, hopefully research will follow.

Roadmap

- Working on implementing the supercompiler described in Bolingbroke [2013].
- (Hopefully) Improving GHC plugin API on the way.
- Collecting benchmark programs - send yours! (with expected optimizations)
- Once we have that, hopefully research will follow.
    - Focus on specific parts(matcher, splitter etc.).

Roadmap

- Working on implementing the supercompiler described in Bolingbroke [2013].
- (Hopefully) Improving GHC plugin API on the way.
- Collecting benchmark programs - send yours! (with expected optimizations)
- Once we have that, hopefully research will follow.
    - Focus on specific parts(matcher, splitter etc.).
    - Work on some of the obvious improvements, like parallelizing the matcher.

Roadmap

- Working on implementing the supercompiler described in Bolingbroke [2013].
- (Hopefully) Improving GHC plugin API on the way.
- Collecting benchmark programs - send yours! (with expected optimizations)
- Once we have that, hopefully research will follow.
  - Focus on specific parts(matcher, splitter etc.).
  - Work on some of the obvious improvements, like parallelizing the matcher.
  - I'm open for more ideas!

References I

M. Bolingbroke and S. Peyton Jones. Supercompilation by Evaluation. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi: 10.1145/1863523.1863540. URL `http://doi.acm.org/10.1145/1863523.1863540`.

M. C. Bolingbroke. Call-by-need supercompilation. Technical Report UCAM-CL-TR-835, University of Cambridge, Computer Laboratory, May 2013. URL `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-835.pdf`.

O. Chitil. Common Subexpression Elimination in a Lazy Functional Language, 1997.

S. L. P. Jones and A. L. Santos. A transformation-based optimiser for Haskell. In *SCIENCE OF COMPUTER PROGRAMMING*, pages 3–47. Elsevier North-Holland, Inc., 1998.

References II

P. A. Jonsson. Time- and Size-Efficient Supercompilation, 201?

I. Klyuchnikov and S. Romanenko. Proving the Equivalence of Higher-Order Terms by Means of Supercompilation. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 193–205. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-11485-4. doi: 10.1007/978-3-642-11486-1_17. URL `http://dx.doi.org/10.1007/978-3-642-11486-1_17`.

M. H. Sørensen, R. Glück, and N. D. Jones. Towards Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. 1994.

V. F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 8:292–325, 1986.

P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.*, 73(2):231–248, Jan. 1988. ISSN 0304-3975. doi: 10.1016/0304-3975(90)90147-A. URL `http://dx.doi.org/10.1016/0304-3975(90)90147-A`.