Rebooting Supercompilation for Haskell

Ömer S. Ağacan
oagacan@indiana.edu

Ryan R. Newton
rrnewton@indiana.edu

August 30, 2015

Rebooting Supercompilation for Haskell - Talk outline

- An overview of supercompilation.

Rebooting Supercompilation for Haskell - Talk outline

- An overview of supercompilation.
- What's interesting about it in the context of Haskell? Current state-of-the-art.

Rebooting Supercompilation for Haskell - Talk outline

- An overview of supercompilation.
- What's interesting about it in the context of Haskell? Current state-of-the-art.
- Overview of how it works.

Rebooting Supercompilation for Haskell - Talk outline

- An overview of supercompilation.
- What's interesting about it in the context of Haskell? Current state-of-the-art.
- Overview of how it works.
- "But where's my supercompiler for Haskell?" My preliminary work and research goals.

Supercompilation: An overview

- Evaluate programs in compile time.

Supercompilation: An overview

- Evaluate programs in compile time.
- Make the most out of known inputs and definitions.

Supercompilation: An overview

- Evaluate programs in compile time.
- Make the most out of known inputs and definitions.
- Evaluate open terms.

Supercompilation in the context of Haskell

- Why is it interesting?
- In a sense, it's the "ultimate" optimization. ("-O99")
- An evaluator-based supercomiler optimizes in the sense that:
  If we have 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, but works well in practice.
  (i.e. if $M < N$ then usually M is a faster program)

Supercompilation in the context of Haskell

It generalizes:

- Deforestation(Wadler [1988])
- Partial evaluation
- Call-pattern specialization(Peyton Jones [2007])
- Ad-hoc optimizations via rewrite rules, e.g. shortcut fusion (Gill et al. [1993]) or library-specific rewrite rules
- "Optimizing SYB is Easy!"(Adams et al. [2014]) and "Optimizing Generics is Easy!"(Magalhães et al. [2010]) style "domain-specific" partial evaluators
- Function specialization(`SPECIALIZE` pragmas)
- ... and many more

Current state-of-the-art for Haskell

- Bolingbroke [2013] shows some great potential:
  - Up to 20*x* faster runtime.
  - Up to 100% reduction in allocation.
- But it also suffers from problems that are inherent to supercompilation:
  - "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)
  - Up to 132*x* compile time.
  - Up to 2.8*x* generated code size.

How it works? An overview

Bolingbroke [2013] laid out a great framework for supercompiling Haskell:

How it works? An overview

Bolingbroke [2013] laid out a great framework for supercompiling Haskell:

- Driving: Take semantics preserving steps. Operational semantics steps, some additional steps like case-of-case transformation(Jones and Santos [1998]).

How it works? An overview

Bolingbroke [2013] laid out a great framework for supercompiling Haskell:

- Driving: Take semantics preserving steps. Operational semantics steps, some additional steps like case-of-case transformation(Jones and Santos [1998]).
- Splitting: When stuck, keep evaluating sub-expressions. Propagate information. After evaluating sub-expressions combine results.

How it works? An overview

Bolingbroke [2013] laid out a great framework for supercompiling Haskell:

- Driving: Take semantics preserving steps. Operational semantics steps, some additional steps like case-of-case transformation(Jones and Santos [1998]).
- Splitting: When stuck, keep evaluating sub-expressions. Propagate information. After evaluating sub-expressions combine results.
- Matching: Evaluating open terms lead to loops. Matcher tries to detect loops, returns information about how to refer to this new loop.

How it works? An overview

Bolingbroke [2013] laid out a great framework for supercompiling Haskell:

- Driving: Take semantics preserving steps. Operational semantics steps, some additional steps like case-of-case transformation(Jones and Santos [1998]).
- Splitting: When stuck, keep evaluating sub-expressions. Propagate information. After evaluating sub-expressions combine results.
- Matching: Evaluating open terms lead to loops. Matcher tries to detect loops, returns information about how to refer to this new loop.
- Termination enforcement: Because perfect matcher is not possible, and some programs just loop.

```
mapOfMap f g = map f . map g

h1 f g a = map f (map g a)

...

h4 f g a =
  case (case a of
          []      -> []
          h1 : t1 -> g h1 : map g t1)  of
    []      -> []
    h0 : t0 -> f h0 : map f t0

...

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

h7 f g a =
  case a of
    []    -> []
    h : t -> f (g h) : h7 f g t
```

Problems with supercompilation operations

Each operation has hard problems to solve.

Problems with supercompilation operations

Each operation has hard problems to solve.

Splitter: (from Bolingbroke [2013])

Problems with supercompilation operations

Each operation has hard problems to solve.

Splitter: (from Bolingbroke [2013])

Propagating too much information may lead to work duplication.

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

```
let b =
      let f = <fib, unrolled a few times>
       in f + 1
    c =
      let f = <fib, unrolled a few times>
       in f + 2
 in (b, c)
```

Problems with supercompilation operations

Each operation has hard problems to solve.

Splitter: (from Bolingbroke [2013])

Propagating too little information may lead to missing optimization opportunities.

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

Problems with supercompilation operations

Each operation has hard problems to solve.

Matcher: Injectivity of substitutions effect optimizations.

Problems with supercompilation operations

Each operation has hard problems to solve.

Matcher: Injectivity of substitutions effect optimizations.

(from Bolingbroke [2013])

```
xor x y = case x of True -> not y; False -> y

goal = (xor a b, xor c c)
```

Problems with supercompilation operations

Each operation has hard problems to solve.

Matcher: Injectivity of substitutions effect optimizations.

(from Bolingbroke [2013])

```
xor x y = case x of True -> not y; False -> y

goal = (xor a b, xor c c)
```

```
xor' x y = case x of True -> not y; False -> y

goal' = (h0 a b, h0 c c)
```

Problems with supercompilation operations

Each operation has hard problems to solve.

Matcher: Injectivity of substitutions effect optimizations.

(from Bolingbroke [2013])

```
xor x y = case x of True -> not y; False -> y

goal = (xor a b, xor c c)
```

```
xor' x y = case x of True -> not y; False -> y

goal' = (h0 a b, h0 c c)
```

```
xor' x y = case x of True -> not y; False -> y

xor'' x = case x of True -> False; False -> False

goal' = (xor' a b, xor'' c)
```

Problems with supercompilation operations

Each operation has hard problems to solve.

Termination enforcement:

Problems with supercompilation operations

Each operation has hard problems to solve.

Termination enforcement:

Some programs just loop.

```
loop n = loop (n + 1)
countFrom n = n : countFrom (n + 1)
```

Problems with supercompilation operations

Each operation has hard problems to solve.

Termination enforcement:

Some programs just loop.

```
loop n = loop (n + 1)
countFrom n = n : countFrom (n + 1)
```

Sometimes detecting loops is not so easy: (growing arguments)

```
reverse_acc []      acc = acc
reverse_acc (h : t) acc = reverse_acc t (h : acc)
goal lst = reverse_acc (reverse_acc lst []) []
...
h_ lst = ... reverse_acc t1 (h1 : []) ...
...
h_ lst = ... reverse_acc t2 (h2 : h1 : []) ...
...
```

"Where's my supercompiler for Haskell?"

- Bolingbroke [2013] has some solutions, and it documents and implements it nicely.
- But we still don't have something that we can use *today*.
- I'm rebooting the supercompiler!
- The goal here is to distribute it as a package, downloadable from Hackage.
- Then the research will follow.

Conclusions

Have a working implementation of supercompiler described in Bolingbroke [2013].

Conclusions

Have a working implementation of supercompiler described in Bolingbroke [2013].

Collecting benchmark programs - send yours! (with expected optimizations)
Create a benchmark suite like Nofib, but for supercompilation-specific problems. (pathological cases, programs with lots of intermediate data structures)

Once we have a working implementation:

- Focus on specific parts(matcher, splitter etc.). Try other ideas from the literature(e.g. homeomorphic embedding for matching)
- Work on some of the obvious improvements, like parallelizing the matcher.
- More experimental ideas:

  Can we formulate it as a search problem and apply ideas from the literature?

  Is profile-driven decision making possible?

  Can we make use of existing rewrite rules mechanism?

  Can we make use of free theorems?

Once we have a working implementation:

- Focus on specific parts(matcher, splitter etc.). Try other ideas from the literature(e.g. homeomorphic embedding for matching and termination enforcement).
- Work on some of the obvious improvements, like parallelizing the matcher.
- More experimental ideas:

  Can we formulate it as a search problem and apply ideas from the literature?
  Is profile-driven decision making possible?
  Can we make use of existing rewrite rules mechanism?
  Can we make use of free theorems?

Thanks!

Github: osa1/sc-plugin IRC: osa1 Mail: oagacan@indiana.edu

References I

M. D. Adams, A. Farmer, and J. P. Magalhães. Optimizing SYB is Easy! In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2619-3. doi: 10.1145/2543728.2543730.

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`.

A. Gill, J. Launchbury, and S. L. Peyton Jones. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X. doi: 10.1145/165180.165214. URL `http://doi.acm.org/10.1145/165180.165214`.

References II

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.

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

J. P. Magalhães, S. Holdermans, J. Jeuring, and A. Löh. Optimizing Generics is Easy! In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '10, pages 33–42, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-727-1. doi: 10.1145/1706356.1706366. URL `http://doi.acm.org/10.1145/1706356.1706366`.

S. Peyton Jones. Call-pattern Specialisation for Haskell Programs. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 327–337, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: 10.1145/1291151.1291200. URL `http://doi.acm.org/10.1145/1291151.1291200`.

References III

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`.