Rebooting Supercompilation for Haskell

Ömer S. Ağacan
oagacan@indiana.edu

Ryan R. Newton
rrnewton@indiana.edu

August 23, 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

- 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. (case expressions)

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 in the context of Haskell

- Why is it interesting?
- In a sense, it's the "ultimate" optimization. ("-O99")
- 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, but works well in practice.

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

- Bolingbroke [2013] shows some great potential:
  - Up to -95.1% runtime improvement.
  - Up to -100.0% allocation improvement.
- 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 $+132002.0\%$ in compile time.
  - Up to $+188.9\%$ in generated code size.

How it works? An overview

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

- Driving: Take steps according to operational semantics. 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 steps according to operational semantics. 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 steps according to operational semantics. 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 tried 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 steps according to operational semantics. 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 tried to detect loops, returns information about how to refer to this new loop.
- Termination checking: Because perfect matcher is not possible, and some programs just loop.

How it works? An overview

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

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

h2 f g a =
  (\f lst -> case lst of
                 []    -> []
                 h : t -> f h : map f t) f (map g a)

h3 f g a =
  case (map g a) of
    []    -> []
    h : t -> f h : map f t
```

How it works? An overview

```
h3 f g a =
  case (map g a) of
    []     -> []
    h : t -> f h : map f t

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

How it works? An overview

```
h4 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])

```
h5 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
```

How it works? An overview

```
h5 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

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

How it works? An overview

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

How it works? An overview

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

```
    map f (map g t)
```

Looks similar to:

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

How it works? An overview

At this point splitter tell us there's a loop.

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

Supercompiled version doesn't generate intermediate list.

Another example, growing arguments

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

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 : []) ...
...
```

Another example, growing arguments

What to do after stopping is completely different story.

Generalization

Rollback

Other/new ideas?

Problems with supercompilation operations

Each operation has hard problems to solve.

Problems with supercompilation operations

Each operation has hard problems to solve.

Problems with supercompilation operations

Each operation has hard problems to solve.

> Splitter: Propagating too much information may lead to work duplication. Propagating too little information may lead to missing optimization opportunities.

Problems with supercompilation operations

Each operation has hard problems to solve.

Splitter: Propagating too much information may lead to work duplication. Propagating too little information may lead to missing optimization opportunities.

Matcher: Matching may lead to work sharing, which may increase memory residency. (Chitil [1997])

Problems with supercompilation operations

Each operation has hard problems to solve.

Splitter: Propagating too much information may lead to work duplication. Propagating too little information may lead to missing optimization opportunities.

Matcher: Matching may lead to work sharing, which may increase memory residency. (Chitil [1997])

Termination checker: Need to be careful with growing arguments.

Problems with supercompilation operations

Each operation has hard problems to solve.

- Splitter: Propagating too much information may lead to work duplication. Propagating too little information may lead to missing optimization opportunities.
- Matcher: Matching may lead to work sharing, which may increase memory residency. (Chitil [1997])
- Termination checker: Need to be careful with growing arguments.

Bolingbroke [2013] has some solutions, and documents it nicely.

Problems with supercompilation operations

Each operation has hard problems to solve.

- Splitter: Propagating too much information may lead to work duplication. Propagating too little information may lead to missing optimization opportunities.
- Matcher: Matching may lead to work sharing, which may increase memory residency. (Chitil [1997])
- Termination checker: Need to be careful with growing arguments.

Bolingbroke [2013] has some solutions, and documents it nicely.

But we still don't have a working implementation.

"Where's my supercompiler for Haskell?"

- I'm rebooting the supercompiler!

"Where's my supercompiler for Haskell?"

- I'm rebooting the supercompiler!
- The goal here is to distribute it as a package, downloadable from Hackage.

"Where's my supercompiler for Haskell?"

- I'm rebooting the supercompiler!
- The goal here is to distribute it as a package, downloadable from Hackage.
- Then the research will follow.

Current status and problems

There has been some changes in GHC:

- Some changes in the Core theory: Roles.
- Lots of refactoring.

GHC API related problems:

- Some needed internals are not exposed by GHC – requires some modifications in GHC.
- 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 is 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 Core definition allows this)

Roadmap

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

Roadmap

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

(Maybe) Improve GHC plugin API on the way.

Roadmap

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

(Maybe) Improve GHC plugin API on the way.

Collecting benchmark programs - send yours! (with expected optimizations)
We can collect something like Nofib, but for supercompilation related problems.

Roadmap

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

(Maybe) Improve GHC plugin API on the way.

Collecting benchmark programs - send yours! (with expected optimizations)
We can collect something like Nofib, but for supercompilation related problems.

Once we have a working implementation:

Roadmap

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

(Maybe) Improve GHC plugin API on the way.

Collecting benchmark programs - send yours! (with expected optimizations)
We can collect something like Nofib, but for supercompilation related problems.

Once we have a working implementation:

- Focus on specific parts(matcher, splitter etc.). Try other ideas from the literature(e.g. homeomorphic embedding).

Roadmap

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

(Maybe) Improve GHC plugin API on the way.

Collecting benchmark programs - send yours! (with expected optimizations)
We can collect something like Nofib, but for supercompilation related problems.

Once we have a working implementation:

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

Roadmap

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

(Maybe) Improve GHC plugin API on the way.

Collecting benchmark programs - send yours! (with expected optimizations)
We can collect something like Nofib, but for supercompilation related problems.

Once we have a working implementation:

- Focus on specific parts(matcher, splitter etc.). Try other ideas from the literature(e.g. homeomorphic embedding).
- Work on some of the obvious improvements, like parallelizing the matcher.
- I'm open for more ideas!

Thanks!

Github: osa1/sc-plugin

IRC: osa1

Email me your slow programs: 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`.

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

References II

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`.

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?

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`.

References III

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`.

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

References IV

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`.