Modular, Higher-Order Cardinality Analysis
in
Theory and Practice

Ilya Sergey  Dmitrios Vytiniotis  Simon Peyton Jones

Microsoft Research

POPL 2014
A story of three program optimisations
Optimisation 1

Which function is better to run?

\[ f_1, f_2 :: [\text{Int}] \rightarrow \text{Int} \]

\[ f_1 \text{ xs} = \text{let } yss = \text{map costly xs} \]
\[ \text{in } \text{squash} \left( \sum_n \text{sum} \left( \text{map} \left( + n \right) yss \right) \right) \]

if invoked more than once by \text{squash}

\[ f_2 \text{ xs} = \text{squash} \left( \sum_n \text{sum} \left( \text{map} \left( + n \right) \left( \text{map costly xs} \right) \right) \right) \]

if invoked at most once by \text{squash}

Better
How many times a function is called?

(call cardinality)
Optimisation 2

“worker-wrapper” split

\[ f \ x = \text{case } x \text{ of } (p, q) \to <\text{cbody}> \]
Optimisation 2

"worker-wrapper" split

"wrapper", usually inlined on-site

\[
f(x) = \text{case } x \text{ of } (p, q) \rightarrow \text{fw } p \quad q
\]

\[
\text{fw } p \quad q = <\text{cbody}>
\]
Optimisation 2

“worker-wrapper” split

What if q is never used in <cbody>?

\[
\begin{align*}
f \ x &= \text{case } x \text{ of } (p, q) \rightarrow \ fw \ p \\
fw \ p &= <cbody>
\end{align*}
\]

Don’t have to pass q to fw!
Which parts of a data structure are certainly not used?

(absence)
 Optimisation 3

smart memoization

f :: Int -> Int -> Int
f x c = if x > 0 then c + 1 else
   if x == 0 then 0 else c - 1

\( \text{g } y = f\ y\ (\text{costly} \ y) \)

Will be used exactly once: no need to memoize!
Which parts of a data structure are used no more than once?

(thunk cardinality)
Cardinality Analysis

- Call cardinality
- Absence
- Thunk cardinality
Usage demands

*(how a value is used)*
Usage demands  
\[ d \ ::= \ C^n(d) | U(d_1^\dagger, d_2^\dagger) | U \]

Cardinality demands  
\[ d^\dagger \ ::= \ A | n*d \]

Usage cardinalities  
\[ n \ ::= \ 1 | \omega \]
<table>
<thead>
<tr>
<th>Usage demands</th>
<th>$d ::= C^n(d) \mid U(d_1^\dagger, d_2^\dagger) \mid U$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality demands</td>
<td>$d^\dagger ::= A \mid n \ast d$</td>
</tr>
<tr>
<td>Usage cardinalities</td>
<td>$n ::= 1 \mid \omega$</td>
</tr>
</tbody>
</table>
Usage demands

\[ d ::= C^n(d) \mid U(d_1^\dagger, d_2^\dagger) \mid U \]

Cardinality demands

\[ d^\dagger ::= A \mid n \ast d \]

Usage cardinalities

\[ n ::= 1 \mid \omega \]
Usage demands

\[ d \ ::= \ C^n(d) \mid U(d_1^\dagger, d_2^\dagger) \mid U \]

Cardinality demands

\[ d^\dagger \ ::= \ A \mid n \ast d \]

Usage cardinalities

\[ n \ ::= \ 1 \mid \omega \]
Usage demands  \[ d \ ::= \ C^n(d) \ | \ U(d_1^\dagger, d_2^\dagger) \ | \ U \]

used at most \( n \) times

Cardinality demands  \[ d^\dagger \ ::= \ A \ | \ n \ast d \]

Usage cardinalities  \[ n \ ::= \ 1 \ | \ \omega \]
Usage Types

*(how a function uses its arguments)*
wurble1 :: \((\omega \cdot U) \rightarrow C^\omega (C^1(U)) \rightarrow \bullet\)

\[ \text{wurble1 } a \ g = g\ 2\ a + g\ 3\ a \]
2.3 Absence analysis

Now we hope that the binding for \( \text{wurble} \)'s right-hand side often does not have a demand signature like this:

\[
\text{wurble} \text{1} :: \omega \times U \rightarrow (C^\omega (C^1(U))) \rightarrow \bullet
\]

\[
\text{wurble} \text{1} \ a \ g = [g]_2 a + [g]_3 a
\]
wurble2 :: \(\omega \times U \rightarrow C^1(C^\omega(U)) \rightarrow \bullet\)

wurble2 a g = sum (map (g \(a\)) [1..1000])
wurble2 :: $\omega \ast U \rightarrow C^1(C^\omega(U)) \rightarrow \bullet$

wurble2 a g = sum (map (g a) [1..1000])
\[ f :: \ 1 \times U(1 \times U, A) \rightarrow \bullet \]

\[ f \ x = \ \text{case } x \ \text{of} \ \ (p, q) \rightarrow p + 1 \]
Usage type depends on a usage context!

(result demand determines argument demands)
Backwards Analysis

Infers demand type basing on a context

\[ P \Downarrow e \downarrow d \Rightarrow \langle \tau ; \varphi \rangle \]
\[ P \mapsto e \downarrow d \Rightarrow \langle \tau ; \varphi \rangle \]

- \( P \) - *signature environment*, maps some of free variables of \( e \) to their demand signatures (*i.e.*, keeps some contextual information)

- \( d \) - *usage demand*, describes the degree to which \( e \) is evaluated

- \( \tau \) - *demand type*, usages that \( e \) places on its arguments

- \( \varphi \) - *fv-usage*, usages that \( e \) places on its free variables
The argument demands `&`, which is defined in Figure 2, and the `†` operator, pronounced "both," is defined in Figure 3. Rule L is shown in Figure 3. Rule P is also shown in Figure 3.

For example, consider the expression `let f = \x.\y. x True in f p q`.

The `†` operator is obtained from the `&` operator by applying the `C` operator, which is defined in Figure 2. Then we can analyse the argument under demand to the least upper bound. We simply analyse the two components, including how many times its sub-components are evaluated, and if we have an uninformative signature.

```
e = \lambda x. \text{case } x \text{ of } (p, q) \rightarrow (p, f \text{ True})
```

```
C^1(U)
```

Here is a lambda, but not good otherwise, for two reasons. Consider the expression `let x = \v. y + v in x 42 + x 239`.

The L side is a lambda, but not good otherwise, for two reasons. Consider the expression `let x = \v. y + v in x 42 + x 239`.

The L side is a lambda, but not good otherwise, for two reasons. Consider the expression `let x = \v. y + v in x 42 + x 239`.

The L side is a lambda, but not good otherwise, for two reasons. Consider the expression `let x = \v. y + v in x 42 + x 239`.
\[ e = \lambda x . \text{case } x \text{ of } (p, q) \to (p, f \text{ True}) \]

\[ \epsilon \vdash e \downarrow C^1(U) \Rightarrow \langle 1 \ast U(\omega \ast U, A) \to \bullet; \{ f \mapsto 1 \ast C^1(U) \} \rangle \]

\[ \tau \]

\[ \psi \]
Each function is a backwards demand transformer. It transforms a context demand to argument demands and fv-demands.
We cannot compute best argument demands for *all* contexts: need to *approximate*.
Demand Lattice

\[ \top = \omega \ast U \]

\[ 1 \ast U \]

\[ 1 \ast U(A, \omega \ast C^1(U)) \]

\[ 1 \ast U(A, A) \]

\[ 1 \ast U(\omega \ast U, A) \]

\[ \omega \ast C^\omega(U(\omega \ast U, A)) \]

\[ \omega \ast C^\omega(U(A, \omega \ast U)) \]

\[ \perp = A \]

\[ 1 \ast C^\omega(U(A, \omega \ast U)) \]
Each function is a monotone backwards demand transformer.
Exploiting demand monotonicity

\[ T = \omega \cdot U \ldots \omega \cdot U \]

Argument demands

Context demand

\[ d_{a_1}^* \ldots d_{a_n}^* \]

\[ d_{a_1}^1 \ldots d_{a_n}^1 \]
Analysis-based annotations
\[ P \vdash e \downarrow d \Rightarrow \langle \tau ; \phi \rangle \]
Elaboration

\[ P \mapsto e \downarrow d \Rightarrow \langle \tau ; \varphi \rangle \rightsquigarrow e \]

- let-bindings in \( e \) are annotated with \( m \in \{0, 1, \omega\} \) to indicate how often the let binding is evaluated;

- Each Lambda \( \lambda^nx.e \) in \( e \) carries an annotation \( n \in \{1, \omega\} \) to indicate how often the lambda is called.
\[\begin{align*}
\epsilon |\downarrow & \quad \text{let } f = \lambda x. \lambda y. x \text{ True in } f \, p \, q \downarrow C^1(U) \\
\Rightarrow & \quad \langle \bullet; \{p \mapsto 1* C^1(U), q \mapsto A\} \rangle \\
\sim & \quad \\
\sim & \quad \text{let } f \overset{1}{=} \lambda^1 x. \lambda^1 y. x \text{ True in } f \, p \, q
\end{align*}\]
Soundness
Restricted operational semantics
(makes sure that the annotations are respected)
Annotating cardinality analysis

produces well-typed programs

annotated programs do not get stuck

Type and effect system

progress and preservation

Restricted operational semantics
Cardinality-enabled optimisations
I. Let-in floating optimisation
let $z \overset{m_1}{=} e_1$ in $(\text{let } f \overset{m_2}{=} \lambda^1 x . e \text{ in } e_2)$
let z \overset{m_1}{=} e_1 \text{ in } (\text{let } f \overset{m_2}{=} \lambda^1 x . e \text{ in } e_2) \\
\implies \text{let } f \overset{m_2}{=} \lambda^1 x . (\text{let } z \overset{m_1}{=} e_1 \text{ in } e) \text{ in } e_2,

for any \ m_1, m_2 \text{ and } z \not\in FV(e_2).
Improvement Theorem 1

Let-in floating *does not* increase the number of execution steps.
2. Smart execution
Optimised CBN Machine

\[ \langle H_1, e_1, S_1 \rangle \implies \ldots \implies \langle H_n, e_n, S_n \rangle \]

- 1-annotated bindings are \textit{not} memoised;
- 0-annotated bindings are \textit{skipped}.
Optimising semantics works faster on elaborated expressions and produces coherent results.
Implementation and Evaluation
The analysis and optimisations are implemented in Glasgow Haskell Compiler (GHC v7.8 and newer): http://github.com/ghc/ghc

- Added 250 LOC to 140 KLOC compiler;
- Runs simultaneously with the strictness analyser;
- Evaluated on
  - nofib benchmark suite,
  - various hackage libraries,
  - the Benchmark Game programs,
  - GHC itself.
### Results on `nofib`

<table>
<thead>
<tr>
<th>Program</th>
<th>Synt. λ¹</th>
<th>Synt. Thnk¹</th>
<th>RT Thnk¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>anna</td>
<td>4.0%</td>
<td>7.2%</td>
<td>2.9%</td>
</tr>
<tr>
<td>bspt</td>
<td>5.0%</td>
<td>15.4%</td>
<td>1.5%</td>
</tr>
<tr>
<td>cacheprof</td>
<td>7.6%</td>
<td>11.9%</td>
<td>5.1%</td>
</tr>
<tr>
<td>calendar</td>
<td>5.7%</td>
<td>0.0%</td>
<td>0.2%</td>
</tr>
<tr>
<td>constraints</td>
<td>2.0%</td>
<td>3.2%</td>
<td>4.5%</td>
</tr>
<tr>
<td>... and 72 more programs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic mean</td>
<td>10.3%</td>
<td>12.6%</td>
<td>5.5%</td>
</tr>
</tbody>
</table>

* as linked and run with libraries
# Results on nofib

The hack (due to A. Gill): hardcode argument cardinalities for `build`, `foldr` and `runST`.

<table>
<thead>
<tr>
<th>Program</th>
<th>Allocs</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No hack</td>
<td>Hack</td>
</tr>
<tr>
<td>anna</td>
<td>-2.1%</td>
<td>-0.2%</td>
</tr>
<tr>
<td>bspt</td>
<td>-2.2%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>cacheprof</td>
<td>-7.9%</td>
<td>-0.6%</td>
</tr>
<tr>
<td>calendar</td>
<td>-9.2%</td>
<td>+0.2%</td>
</tr>
<tr>
<td>constraints</td>
<td>-0.9%</td>
<td>-0.0%</td>
</tr>
<tr>
<td>... and 72 more programs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Min     | -95.5%  | -10.9% | -28.2%  | -12.1% |
| Max     | +3.5%   | +0.5%  | +1.8%   | +2.8%  |
| Geometric mean | -6.0%  | -0.3%  | -2.2%   | -1.4%  |
Compiling with optimised GHC

• We compiled GHC itself with cardinality optimisations;

• Then we measured improvement in *compilation runtimes*.

---

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>GHC Alloc Δ</th>
<th>GHC RT Δ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>No hack</td>
<td>Hack</td>
</tr>
<tr>
<td>anna</td>
<td>5740</td>
<td>-1.6%</td>
<td>-1.5%</td>
</tr>
<tr>
<td>cacheprof</td>
<td>1600</td>
<td>-1.7%</td>
<td>-0.4%</td>
</tr>
<tr>
<td>fluid</td>
<td>1579</td>
<td>-1.9%</td>
<td>-1.9%</td>
</tr>
<tr>
<td>gamteb</td>
<td>1933</td>
<td>-0.5%</td>
<td>-0.1%</td>
</tr>
<tr>
<td>parser</td>
<td>2379</td>
<td>-0.7%</td>
<td>-0.2%</td>
</tr>
<tr>
<td>veritas</td>
<td>4674</td>
<td>-1.4%</td>
<td>-0.3%</td>
</tr>
</tbody>
</table>
To take away

• **Cardinality analysis** is *simple* to design and understand: it’s all about *usage demands* and *demand transformers*;

• It is **cheap to implement**: we added only 250 LOC to GHC;

• It is conservative, which makes it **fast** and **modular**;

• *Call demands* make it **higher-order**, so the analysis can infer demands on higher-order function arguments;

• It is **reasonably efficient**: optimised GHC compiles up to 4% faster.

Thanks!