Partial Graph Reduction:
A New Optimization Technique for Higher-Order Programs

Lionel Parreaux
January 8, 2020
INRIA Paris
Inlining in Optimizing Compilers
Consider this program:

```ocaml
let f x = x + 7
in f 3 * f 4
```
Consider this program:

```ocaml
let f x = x + 7
in f 3 * f 4
```

An optimizing compiler will *inline* `f`, giving:

\[(3 + 7) \times (4 + 7)\]
Consider this program:

```ocaml
let f x = x + 7
in f 3 * f 4
```

An optimizing compiler will inline $f$, giving:

$$(3 + 7) \times (4 + 7)$$

Exposing constant folding optimization; resulting in:

110
Question: how to optimize this (Haskell) program?

```haskell
let f x =
  let z = E3\text{isJust } x \rangle
  in E0\langle \text{case } x \text{ of}
    \text{Just } a \rightarrow E1\langle a, z \rangle
    \text{Nothing} \rightarrow E2\langle z \rangle \rangle
  in f (\text{Just } 2) + f \text{ Nothing}
```
Problem

Original program:

```
let f x =
  let z = E3⟨isJust x⟩
  in E0⟨case x of
    Just a → E1⟨a, z⟩
    Nothing → E2⟨z⟩⟩
in f (Just 2) + f Nothing
```

After inlining:

```
let f x =
  let z = E3⟨isJust x⟩
  in E0⟨case x of
    Just a → E1⟨a, z⟩
    Nothing → E2⟨z⟩⟩
in
  (let z0 = E3⟨isJust (Just 2)⟩
   in E0⟨case Just 2 of
     Just a → E1⟨a, z0⟩
     Nothing → E2⟨z0⟩⟩
   ) +
  (let z1 = E3⟨isJust (Nothing)⟩
   in E0⟨case Nothing of
     Just a → E1⟨a, z1⟩
     Nothing → E2⟨z1⟩⟩
  )
```
Problem

Original program:

```ocaml
let f x =  
  let z = E3⟨isJust x⟩  
  in E0⟨case x of  
    Just a → E1⟨a, z⟩  
    Nothing → E2⟨z⟩ ⟩ 
in f ⟨Just 2⟩ + f ⟨Nothing⟩
```

After reduction:

```ocaml
let f x =  
  let z = E3⟨isJust x⟩  
  in E0⟨case x of  
    Just a → E1⟨a, z⟩  
    Nothing → E2⟨z⟩ ⟩ 
in (let z0 = E3⟨true⟩  
  in E0⟨E1⟨2 + z0⟩ ⟩) +  
  (let z1 = E3⟨false⟩  
  in E0⟨E2⟨z1⟩ ⟩)
```
Problem

Original program:

```ml
let f x =  
    let z = E3⟨isJust x⟩  
    in E0⟨case x of  
        Just a  → E1⟨a, z⟩  
        Nothing  → E2⟨z⟩ ⟩  
    in f (Just 2) + f Nothing
```

After dead code elimination:

```ml
(let z0 = E3⟨True⟩  
    in E0⟨E1⟨2 + z0⟩⟩)  
+  
(let z1 = E3⟨False⟩  
    in E0⟨E2⟨z1⟩⟩)
```
Problem

Original program:

```haskell
let f x =
    let z = E3⟨isJust x⟩
    in E0⟨case x of
        Just a → E1⟨a, z⟩
        Nothing → E2⟨z⟩ ⟩
    in f (Just 2) + f Nothing
```

After dead code elimination:

```haskell
(let z0 = E3⟨True⟩
    in E0⟨E1⟨2 + z0⟩⟩)
+ (let z1 = E3⟨False⟩
    in E0⟨E2⟨z1⟩⟩)
```

**Problem:** Duplication!
Problem

Original program:

```ml
let f x =
  let z = E3⟨isJust x⟩
  in E0⟨case x of
    Just a → E1⟨a, z⟩
    Nothing → E2⟨z⟩ ⟩
in f (Just 2) + f Nothing
```

After dead code elimination:

```ml
(let z0 = E3⟨True⟩
in E0⟨E1⟨2 + z0⟩⟩) +
(let z1 = E3⟨False⟩
in E0⟨E2⟨z1⟩⟩)
```

Problem: Duplication!

What we would really like:

```ml
let f0 x0 = E0⟨x0⟩ in let f1 x1 = E3⟨x1⟩ in
f0 (E1⟨2, f1 False⟩) + f0 (E2⟨f1 True⟩)
```
Problems of Inlining

Traditional inlining:

• needs heuristics to avoid code explosion
• causes code duplication (loss of sharing)
• can’t handle optimization across recursive calls

Underlying problem: inlining is *all-or-nothing*. 
A Graph-Based Approach for Partial/Incremental Inlining
Ideas:

- Represent functional programs as graphs
- Use special nodes to encode sharing contexts
- Adapt the graphs to expose optimizations, without duplicating entire function bodies
- Reconstruct functional programs at the end
A Graph-Based Approach for Partial Inlining

Ideas:

- Represent functional programs as graphs
- Use special nodes to encode sharing contexts
- Adapt the graphs to expose optimizations, without duplicating entire function bodies
- Reconstruct functional programs at the end

Generalizes several existing optimizations.
Original program:

```ml
let f x =
  let z = E3⟨isJust x⟩
  in E0⟨case x of
    Just a  → E1⟨a, z⟩
    Nothing → E2⟨z⟩⟩
  in f (Just 2) + f Nothing
```

Beta Reduction Without Copying

\[ \lambda x. u_0 \beta (\text{non-copying}) \]

\[ \lambda x. u_1 \beta (\text{copying}) \]
Beta Reduction Without Copying

\[ \text{\(\lambda x.\)} \]

\[ b \]

\[ x \]

\[ a \]

\[ u_0 \]

\[ \beta \]

(copied)

\[ u_1 \]

\[ \beta \]

(non-copied)

\[ u_0 \]

\[ \lambda x.\]
Motivating Example: Beta Reduction

\[ \lambda x. E0 \]

\[ Nothing \]

\[ Just \ 2 \]

\[ E0 \]

\[ Nothing \]

\[ Just \ 2 \]

\[ u_1 \]

\[ \alpha \uparrow \]

\[ u_0 \]

\[ \lambda x. \]

\[ E0 \]

\[ Nothing \]

\[ case \]

\[ Just \]

\[ Nothing \]

\[ E1 \]

\[ E2 \]

\[ Just \#1 \]

\[ E3' \]

\[ \alpha? \]

\[ u_2 \]

\[ \emptyset \]

\[ u_1 \]

\[ Just \ 2 \]

\[ x \]
Commuting and Reducing Copy Nodes

Copying applications

Moreover, copy nodes annihilate with drop nodes: $[\alpha^\uparrow u_0 \Rightarrow u_1]$
Commuting and Reducing Copy Nodes

Copying applications

Resolving branches
Commuting and Reducing Copy Nodes

Copying applications

\[ u_0 \alpha \uparrow \quad \Rightarrow \quad \emptyset \quad \Rightarrow \quad u_1 \]

copy/app commuting

Resolving branches

\[ u_0 \alpha \uparrow \quad \xleftarrow{\text{branch red.}} \quad (\alpha = \alpha') \quad u_0 \alpha \uparrow \quad \xrightarrow{\alpha' \uparrow \text{?}} \quad \text{yes} \quad \text{no} \quad \xrightarrow{\alpha \neq \alpha'} \quad u_1 \]

Moreover, copy nodes annihilate with drop nodes: \([\alpha \uparrow] [\emptyset] u \rightarrow u\)
Optimizing Across Function Call Boundaries

Pushing copy nodes down:

![Diagram showing the process of pushing copy nodes down with type nodes and alpha operators.]

- $\text{isJust} \xrightarrow{\alpha \uparrow} \text{Just} \rightarrow \ldots$
- Exposing a redex via commuting

- $\text{isJust} \rightarrow \text{Just} \rightarrow \alpha \uparrow \rightarrow \ldots$
Optimizing Across Function Call Boundaries

Pushing copy nodes down:

Pulling branch nodes up:
Motivating Example: Reducing

\[
\begin{align*}
&\alpha \uparrow @ \\
&\lambda x. \\
&E_0 \quad \text{Nothing} \\
&\alpha? \quad \text{case} \quad \text{Just} \quad \text{Nothing} \\
&\text{case} \quad \text{Just} \quad \text{Nothing} \\
&E_1 \quad \text{E2} \\
&\alpha? \quad \text{E3'} \\
&\text{Just#1} \quad \text{Just#1} \\
&\text{Just 2} \\
&\times \\
\end{align*}
\]
Scopes and Variable Capture

\[ f_1 = \lambda x. (\lambda y. x + y) (x + 1) \]
Scopes and Variable Capture

$f_1 = \lambda x. (\lambda y. x + y) (x + 1)$
Scopes and Variable Capture

\[ f_1 = \lambda x. (\lambda y. x + y) (x + 1) \]

Uses “stop” nodes \([\downarrow]\) to delimit scopes.
Control nodes \([i]\) are: “copy” \([\alpha\uparrow]\), “drop” \([\emptyset]\), “stop” \([\downarrow]\)
More commuting for control nodes

Control nodes \( i \) are: “copy” \( \alpha \uparrow \), “drop” \( \emptyset \), “stop” \( \downarrow \)

Copy nodes can be parameterized by a control node instruction \( \alpha? \):
More commuting for control nodes

Control nodes \([i]\) are: “copy” \([\alpha^\uparrow]\) , “drop” \([\emptyset]\) , “stop” \([\downarrow]\)

Copy nodes can be parameterized by a control node instruction \(i\):
Commuting control noes across lambdas

Instruction parameter introduced when commuting with lambda:

```
\[\alpha \uparrow i\] releases \[i\] when meeting \[\downarrow\]; drops it when meeting \[\emptyset\].
```
Commuting control noes across lambdas

Instruction parameter introduced when commuting with lambda:

\[ [\alpha \uparrow [i]] \text{ releases } i \text{ when meeting } [\downarrow] \text{; drops it when meeting } [\emptyset]. \]
Properties of PGR

IGR formalized as $\lambda\{\mapsto\}$.

**Theorem (Small step rewrites preserve semantics)**
Reduction defined in $\lambda\{\mapsto\}$ is no stronger than strong reduction in $\lambda$ calculus: if $P_0 \rightarrow P_1$ with $P_0 \\mathcal{WS}$, then $U[P_0] \equiv U[P_1]$.

**Theorem (Exhaustiveness of Reduction)**
$U[\cdot]$ is a simulation: if $U[P_0] \rightarrow e_1$ then there exists a $P_1$ such that $P_0 \rightarrow^* P_1$ and $U[P_1] = e_1$.

**Theorem (Maximal Sharing)**
We do not duplicate applications: in a program’s graph after rewriting, there will be at most as many applications as in the original program.
IGR formalized as $\lambda\{\mapsto\}$.

**Theorem (Small step rewrites preserve semantics)**
Reduction defined in $\lambda\{\mapsto\}$ is no stronger than strong reduction in $\lambda$ calculus: if $P_0 \xrightarrow{} P_1$ with $P_0 \ \text{WS}$, then $U[P_0] \equiv U[P_1]$.

**Theorem (Exhaustiveness of Reduction)**
$U[\cdot]$ is a simulation: if $U[P_0] \xrightarrow{} e_1$ then there exists a $P_1$ such that $P_0 \xrightarrow{}* P_1$ and $U[P_1] = e_1$.

**Theorem (Maximal Sharing)**
We do not duplicate applications: in a program’s graph after rewriting, there will be at most as many applications as in the original program.

Incidental result: IGR is a $\beta$-optimal evaluator
Ideas:

• each copy identifier denotes a *scope*, in which runtime work is shared
  • copy node: function return
  • drop node: function parameter
  • stop node: variable capture

• reconstitute scopes as corresponding functions

• branches that cannot be solved locally use a flag
  — consider: $[[\emptyset] [\emptyset]]\alpha?...$

• use *undefined* when no argument make sense
Scheduling

Ideas:

- each copy identifier denotes a *scope*, in which runtime work is shared
  - copy node: function return
  - drop node: function parameter
  - stop node: variable capture

- reconstitute scopes as corresponding functions

- branches that cannot be solved locally use a flag — consider: $[[\emptyset] \ [\emptyset]]\alpha\ldots$

- use *undefined* when no argument make sense

Example: $f \ a = \text{let } \ tmp = g \ a \ \text{in } (\ tmp + 1, \ tmp - 1)$

with usage: $\text{case } f \ a \ \text{of } (u, v) \rightarrow u + v$
Motivating Example: Scheduling

After scheduling:

\[
\begin{align*}
\text{let } & f_0 \ x_0 = E_0(x_0) \text{ in} \\
\text{let } & f_1 \ x_1 = E_3(x_1) \text{ in} \\
& f_0 \ (E_1(2, f_1 \text{ False})) \\
& + f_0 \ (E_2(f_1 \text{ True}))
\end{align*}
\]
Enabled Optimizations

Generalized optimization techniques:

- Function outlining, partial inlining
- Uncurrying and efficient multiple returns
- Call-pattern specialisation
- Return-pattern specialisation (new)
Enabled Optimizations

Generalized optimization techniques:

- Function outlining, partial inlining
- Uncurrying and efficient multiple returns
- Call-pattern specialisation
- Return-pattern specialisation (new)

A new approach to:

- Online partial evaluation
- Rewrite rule application
- Handling of join points (immediate or “obvious” in the graph)
- Lambda lifting and defunctionalization
- Deforestation
After reductions, $P$ and $Q$ have \textit{equivalent} PGR representations:

\begin{verbatim}
P: let f x y = x : f y x  
    in ... f a b ... f c d ...
Q: let f (x, y) = x : f (y, x)  
    in ... f (a, b) ... f (c, d) ...
\end{verbatim}
Uncurrying and efficient multiple returns

After reductions, \( P \) and \( Q \) have equivalent PGR representations:

\[
\begin{align*}
P: \ & \text{let } f \ x \ y = x : f \ y \ x \\
& \quad \text{in } ... f \ a \ b ... f \ c \ d ... \\
Q: \ & \text{let } f \ (x, \ y) = x : f \ (y, \ x) \\
& \quad \text{in } ... f \ (a, \ b) ... f \ (c, \ d) ... \\
\end{align*}
\]

Use the most efficient implementation of argument-passing available — in Haskell, unboxed tuples:

\[
\begin{align*}
\text{let } & f \ (# x, \ y #) = x : f \ (# y, \ x #) \\
& \quad \text{in } ... f \ (# a, \ b #) ... f \ (# c, \ d #) ... \\
\end{align*}
\]
Out of the box: optimize across recursive calls:

```haskell
maxMaybe [] = Nothing
maxMaybe (x : xs) = case maxMaybe xs of
    Just m → Just (if x > m then x else m)
    Nothing → Just x
```
Out of the box: optimize across recursive calls:

\[
\begin{align*}
\text{maxMaybe} \; [] &= \text{Nothing} \\
\text{maxMaybe} \; (\text{x} : \text{xs}) &= \text{case} \; \text{maxMaybe} \; \text{xs} \; \text{of} \\
& \quad \text{Just} \; \text{m} \; \rightarrow \; \text{Just} \; (\text{if} \; \text{x} > \text{m} \; \text{then} \; \text{x} \; \text{else} \; \text{m}) \\
& \quad \text{Nothing} \; \rightarrow \; \text{Just} \; \text{x}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Program name</th>
<th>GHC</th>
<th>PGR + GHC</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxMaybe</td>
<td>136.0 (6.176)</td>
<td>33.41 (3.297)</td>
</tr>
</tbody>
</table>

(All optimized with GHC -O3.)
Online Partial Evaluation

Uses recursion markers; allows reducing recursive functions with non-recursive subgraphs

\[
\text{max3 } x \ y \ z = \text{fromJust} \ (\text{maxMaybe} \ [x, \ y, \ z])
\]

Optimized to:

\[
\text{max3 } x \ y \ z = \\
\text{let } c = \text{case } y > z \text{ of } \{\text{True } \rightarrow y; \text{False } \rightarrow z\} \\
\text{in case } x > c \text{ of } \{\text{True } \rightarrow x; \text{False } \rightarrow c\}
\]
Uses recursion markers; allows reducing recursive functions with non-recursive subgraphs

\[
\text{max3 } x \ y \ z = \text{fromJust} \ (\text{maxMaybe } [x, y, z])
\]

Optimized to:

\[
\text{max3 } x \ y \ z = \\
\quad \text{let } c = \text{case } y > z \text{ of } \{ \text{True } \rightarrow y; \text{False } \rightarrow z \} \\
\quad \text{in case } x > c \text{ of } \{ \text{True } \rightarrow x; \text{False } \rightarrow c \}
\]

<table>
<thead>
<tr>
<th>Program name</th>
<th>GHC</th>
<th>PGR + GHC</th>
</tr>
</thead>
<tbody>
<tr>
<td>max3</td>
<td>52.49 (1.039)</td>
<td>29.23 (0.191)</td>
</tr>
</tbody>
</table>
Partial graph reduction (PGR) makes inlining not “all-or-nothing”

Generalizes and facilitates existing optimizations, making them more robust (no heuristics)

Uses context sharing, similar to optimal reduction

(but cannot be expressed with interaction nets due to some commutings)