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

Lionel Parreaux

January 8, 2020

INRIA Paris

Inlining in Optimizing Compilers

Basics of inlining

Consider this program:

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

Basics of inlining

Consider this program:

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

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

```
(3 + 7) * (4 + 7)
```

Basics of inlining

Consider this program:

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

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

```
(3 + 7) * (4 + 7)
```

Exposing constant folding optimization; resulting in:

```
110
```

Problem

Question: how to optimize this (Haskell) 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
```

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:

```
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:

```
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:

```
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:

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

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 dead code elimination:

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

Problem: Duplication!

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 dead code elimination:

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

Problem: Duplication!

What we would really like:

```
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

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

A Graph-Based Approach for Partial Inlining

Ideas:

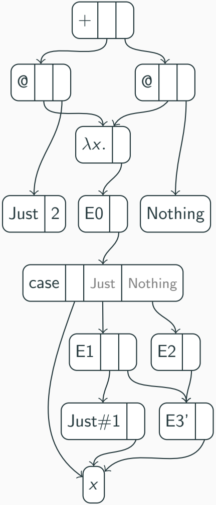
- Represent functional programs as graphs
- Use special nodes to encode sharing *contexts*
- Adapt the graphs to expose optimizations, without duplicating entire function bodies
- Reconstruct functional programs at the end

Generalizes several existing optimizations.

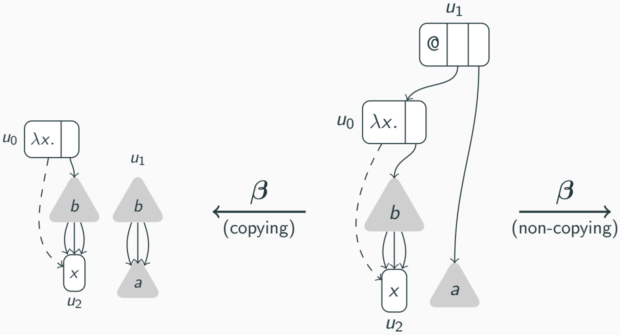
Functional Programs as Graphs

Original program:

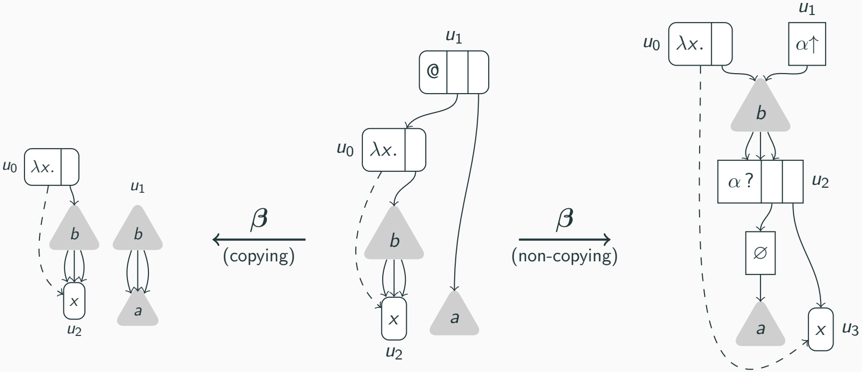
```
let f x =
  let z = E3⟨isJust x⟩
  in E0⟨ case x of
    Just a    →  E1⟨a, z⟩
    Nothing   →  E2⟨z⟩ ⟩
in f (Just 2) + f Nothing
```



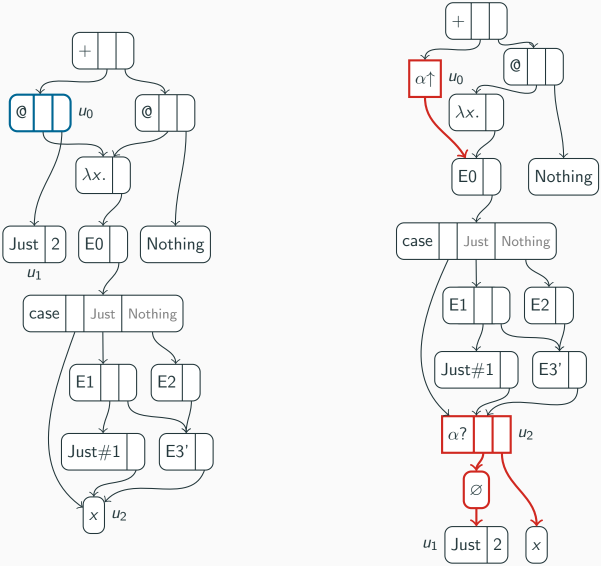
Beta Reduction Without Copying



Beta Reduction Without Copying

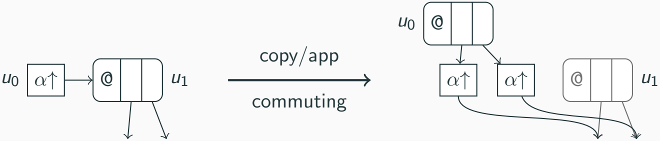


Motivating Example: Beta Reduction



Commuting and Reducing Copy Nodes

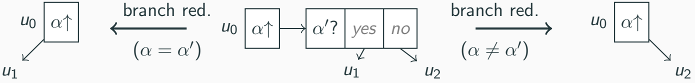
Copying applications



Commuting and Reducing Copy Nodes

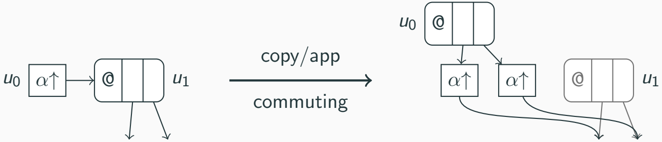
Copying applications

Resolving branches

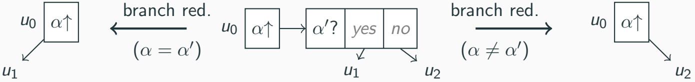


Commuting and Reducing Copy Nodes

Copying applications



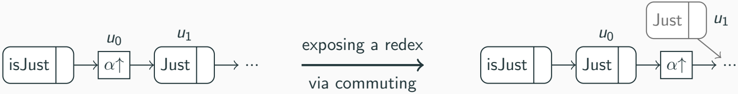
Resolving branches



Moreover, copy nodes annihilate with drop nodes: $[\alpha\uparrow]\ [\varnothing]\ u \to u$

Optimizing Across Function Call Boundaries

Pushing copy nodes down:



u_0 u_1

isJust → $\alpha\uparrow$ → Just → ⋯

exposing a redex

via commuting

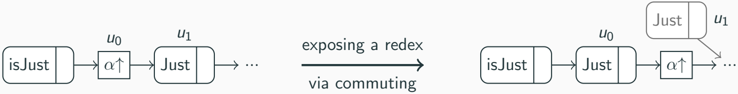
Just u_1

u_0

isJust → Just → $\alpha\uparrow$ → ⋯

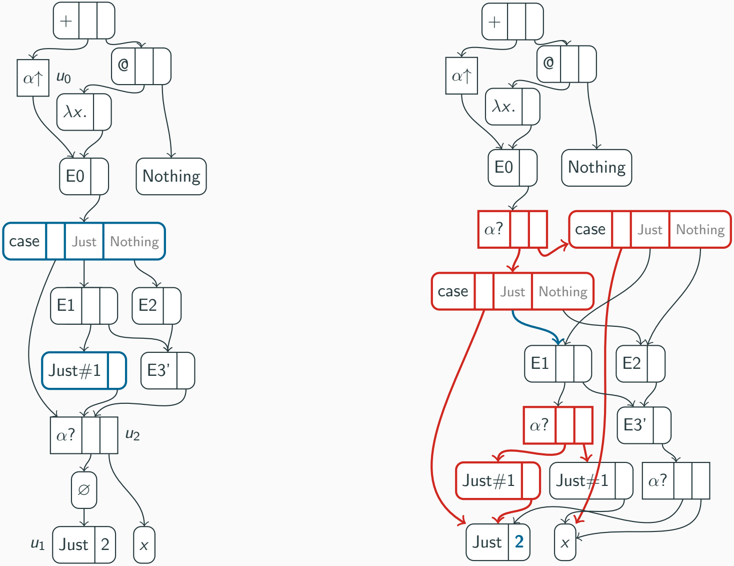
Optimizing Across Function Call Boundaries

Pushing copy nodes down:

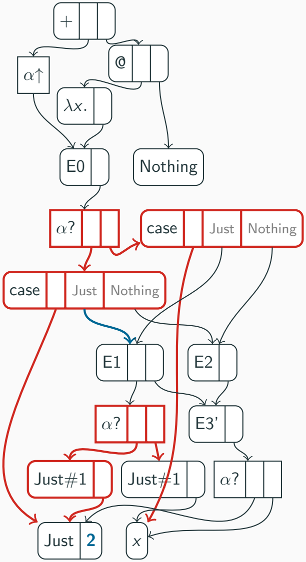


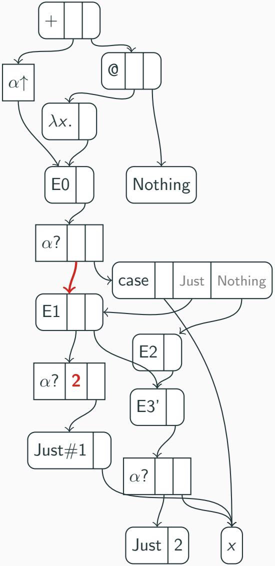
Pulling branch nodes up:

Motivating Example: Commuting



Motivating Example: Reducing

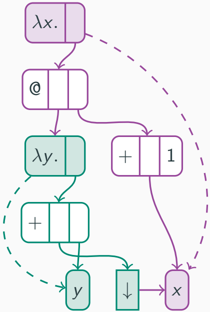




Scopes and Variable Capture

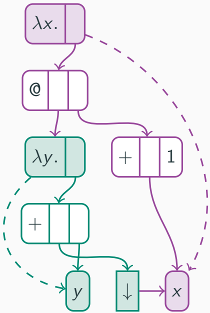
$$f_1 = \lambda x.\ (\lambda y.\ x + y)\ (x + 1)$$

Scopes and Variable Capture



$$f_1 = \lambda x.\ (\lambda y.\ x + y)\ (x + 1)$$

Scopes and Variable Capture



$$f_1 = \lambda x.\ (\lambda y.\ x + y)\ (x + 1)$$

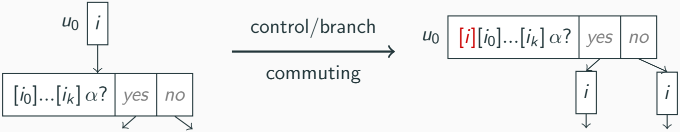
Uses “stop” nodes [↓] to delimit scopes.

More commuting for control nodes

Control nodes $[\,i\,]$ are: “copy” $[\alpha{\uparrow}]$, “drop” $[\varnothing]$, “stop” $[{\downarrow}]$

More commuting for control nodes

Control nodes $[\,i\,]$ are: "copy" $[\alpha\uparrow]$, "drop" $[\varnothing]$, "stop" $[\downarrow]$



u_0 i

$[i_0]\ldots[i_k]\,\alpha?$ *yes* *no*

control/branch

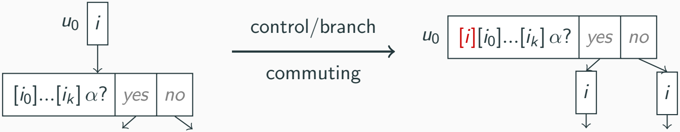
commuting

u_0 $[i][i_0]\ldots[i_k]\,\alpha?$ *yes* *no*

i i

More commuting for control nodes

Control nodes $[\,i\,]$ are: “copy” $[\alpha\uparrow]$, “drop” $[\varnothing]$, “stop” $[\downarrow]$



u_0 i

$[i_0]\ldots[i_k]\,\alpha?$ *yes* *no*

control/branch

commuting

u_0 $[i][i_0]\ldots[i_k]\,\alpha?$ *yes* *no*

i i

Copy nodes can be parameterized by a control node instruction *i*:

u_3 $\alpha\uparrow$ i

$\downarrow$

u_4

payload-copy/stop

reduction

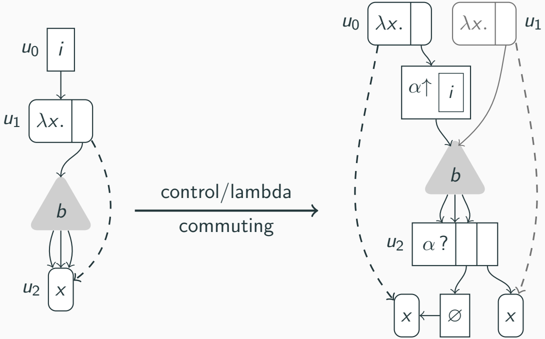
u_3 $\downarrow$

i

u_4

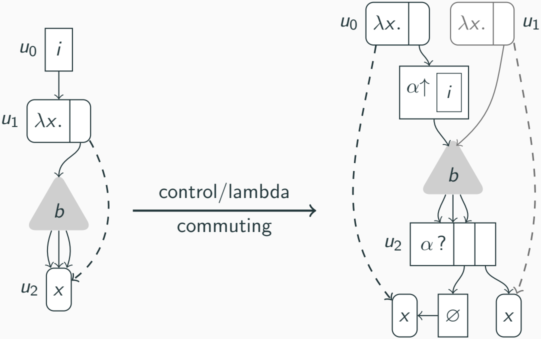
Commuting control noes across lambdas

Instruction parameter introduced when commuting with lambda:



Commuting control noes across lambdas

Instruction parameter introduced when commuting with lambda:



$[\alpha\uparrow[i]]$ releases i when meeting $[\downarrow]$; drops it when meeting $[\varnothing]$.

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* λ *calculus: if* $\mathbf{P}_0 \to \mathbf{P}_1$ *with* $\mathbf{P}_0$ $\mathcal{WS}$*, then* $\mathcal{U}[\![\mathbf{P}_0]\!] \equiv \mathcal{U}[\![\mathbf{P}_1]\!]$.

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

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.

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* λ *calculus: if* $\mathbf{P}_0 \to \mathbf{P}_1$ *with* $\mathbf{P}_0$ $\mathcal{WS}$*, then* $\mathcal{U}[\![\mathbf{P}_0]\!] \equiv \mathcal{U}[\![\mathbf{P}_1]\!]$*.*

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

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 β-optimal evaluator

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: $[[\varnothing]\ [\varnothing]\,]\alpha?\ldots$
- use *undefined* when no argument make sense

Scheduling

Ideas:

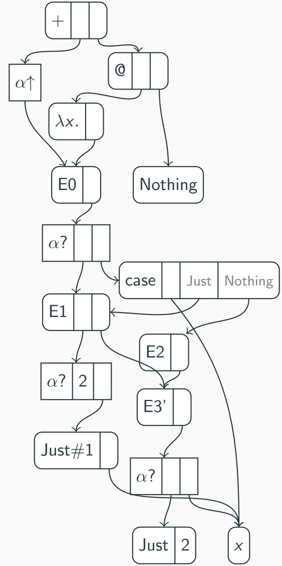
- each copy identifier denotes a *scope*, in which runtime work is shared
  - copy node: function return
  - drop node: function parameter
  - stop node: variable capture
- reconstitute scopes as corresponding functions
- branches that cannot be solved locally use a flag

  — consider: $[[\varnothing]\ [\varnothing]]\alpha$?...
- use *undefined* when no argument make sense

Example: `f a = let tmp = g a in (tmp + 1, tmp - 1)`

with usage: `case f a of (u,v) → u + v`

Motivating Example: Scheduling



After scheduling:

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

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

Uncurrying and efficient multiple returns

After reductions, `P` and `Q` have *equivalent* PGR representations:

```
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) ...
```

Uncurrying and efficient multiple returns

After reductions, `P` and `Q` have *equivalent* PGR representations:

```
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) ...
```

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

```
let f (# x, y #) = x : f (# y, x #)
 in ... f (# a, b #) ... f (# c, d #) ...
```

Return-pattern Specialisation

Out of the box: optimize across recursive calls:

```
maxMaybe [] = Nothing
maxMaybe (x : xs) = case maxMaybe xs of
  Just m  →  Just (if x > m then x else m)
  Nothing  →  Just x
```

Return-pattern Specialisation

Out of the box: optimize across recursive calls:

```
maxMaybe [] = Nothing
maxMaybe (x : xs) = case maxMaybe xs of
  Just m  →  Just (if x > m then x else m)
  Nothing  →  Just x
```

Program name	GHC	PGR + GHC
maxMaybe	136.0 (6.176)	33.41 (3.297)

(All optimized with GHC -O3.)

Online Partial Evaluation

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

```
max3 x y z = fromJust (maxMaybe [x, y, z])
```

Optimized to:

```
max3 x y z =
 let c = case y > z of {True → y; False → z}
  in case x > c of {True → x; False → c}
```

Online Partial Evaluation

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

```
max3 x y z = fromJust (maxMaybe [x, y, z])
```

Optimized to:

```
max3 x y z =
 let c = case y > z of {True → y; False → z}
  in case x > c of {True → x; False → c}
```

Program name	GHC	PGR + GHC
max3	52.49 (1.039)	29.23 (0.191)

Conclusions

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)