Waddle

Always-Canonical Intermediate Representation

Eric Fritz

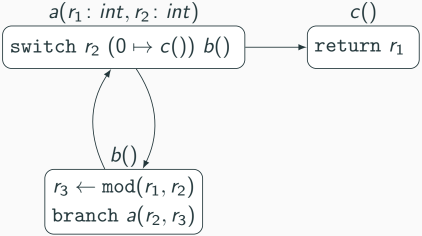
December 3, 2018

University of Wisconsin – Milwaukee

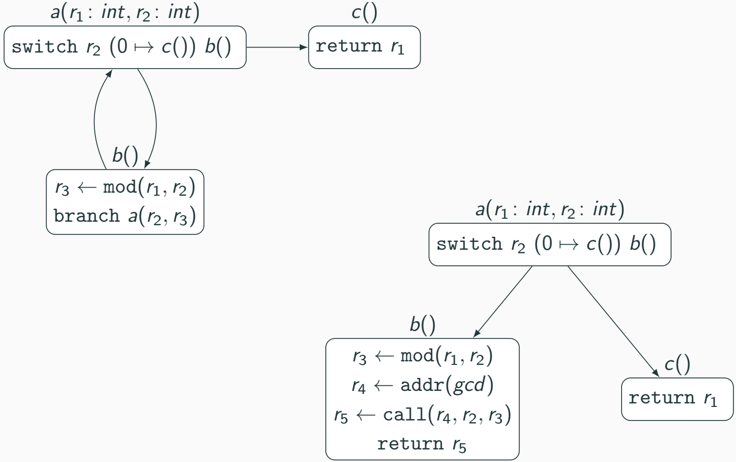
Standard Compiler Architecture

Frontend: lex, parse, name resolution, typechecking
Middle-end: high-level symbolic optimization
Backend: machine-level optimization, register assignment, synthesis

Waddle's IR: Euclid's Algorithm



Waddle's IR: Euclid's Algorithm



Non-Incremental Architecture: Pass Manager

For each optimization o (in a fixed order) and for each function f:

Non-Incremental Architecture: Pass Manager

For each optimization o (in a fixed order) and for each function f:
Recalculate all *dirty* structures/properties of f required by o

Non-Incremental Architecture: Pass Manager

For each optimization o (in a fixed order) and for each function f:
Recalculate all *dirty* structures/properties of f required by o
Execute o over f

Non-Incremental Architecture: Pass Manager

For each optimization o (in a fixed order) and for each function f:

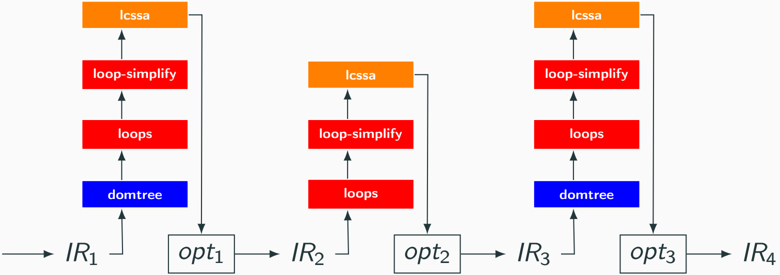
Recalculate all *dirty* structures/properties of f required by o

Execute o over f

Mark all structures/properties of f dirty unless explicitly preserved by o

Non-Incremental Architecture: Pass Manager

For each optimization o (in a fixed order) and for each function f:
Recalculate all *dirty* structures/properties of f required by o
Execute o over f
Mark all structures/properties of f dirty unless explicitly preserved by o



Waddle's Architecture: Always-Canonical

For each optimization o_C (in a fixed order) and for each function f:

Waddle's Architecture: Always-Canonical

For each optimization o_C (in a fixed order) and for each function f:
Execute o_C over f

Waddle's Architecture: Always-Canonical

For each optimization o_C (in a fixed order) and for each function f:
Execute o_C over f

(o_C is written to incrementally maintain common structures/properties)

Dream Architecture: Phaseless

For each function f:

Dream Architecture: Phaseless

For each function f:
Build worklist of optimization opportunities by benefit

Dream Architecture: Phaseless

For each function f:
Build worklist of optimization opportunities by benefit

While most beneficial optimization o is above threshold,
Dequeue and execute o

Dream Architecture: Phaseless

For each function f:
Build worklist of optimization opportunities by benefit

While most beneficial optimization o is above threshold,
Dequeue and execute o

As o modifies the program,
new opportunities are scored and enqueued

What Does Waddle Maintain? (1)

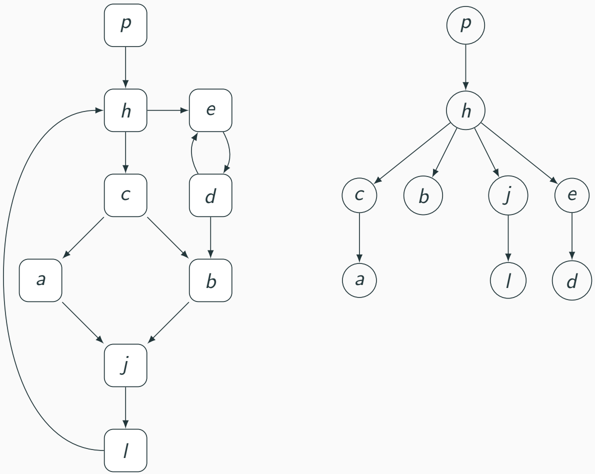
Dominator Tree

encodes which blocks occur on all paths to another block

What Does Waddle Maintain? (1)

Dominator Tree

encodes which blocks occur on all paths to another block



What Does Waddle Maintain? (2)

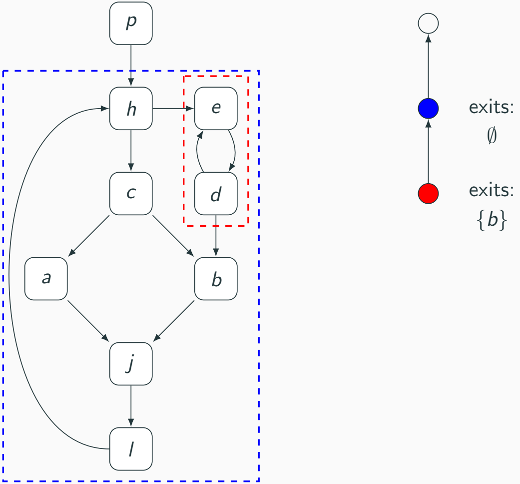
Loop Nesting Forest

encodes loop body sets · loop exit sets · loop nesting structure

What Does Waddle Maintain? (2)

Loop Nesting Forest

encodes loop body sets · loop exit sets · loop nesting structure



What Does Waddle Maintain? (3)

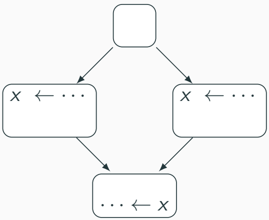
SSA Form

all names defined once

What Does Waddle Maintain? (3)

SSA Form

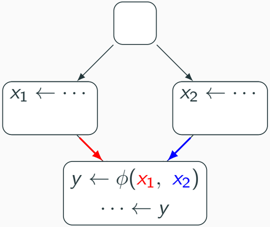
all names defined once



What Does Waddle Maintain? (3)

SSA Form

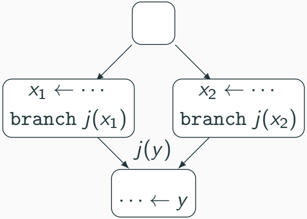
all names defined once



What Does Waddle Maintain? (3)

SSA Form

all names defined once



What Does Waddle Maintain? (4)

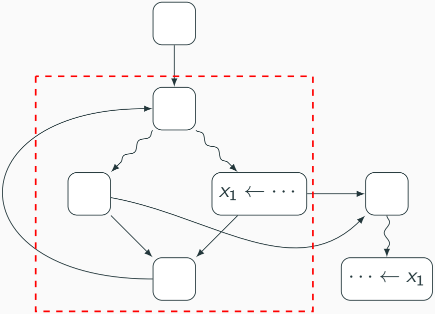
LCSSA Form

all uses of name occur within defining loop

What Does Waddle Maintain? (4)

LCSSA Form

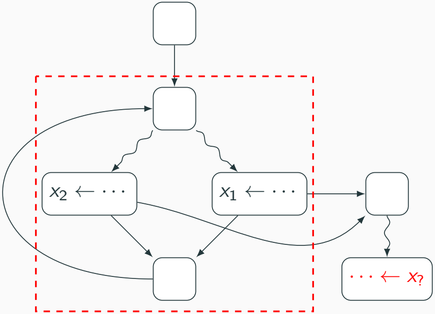
all uses of name occur within defining loop



What Does Waddle Maintain? (4)

LCSSA Form

all uses of name occur within defining loop



What Does Waddle Maintain? (5)

'Canonical' Properties

Equivalent to LLVM's Loop Simplify Form

What Does Waddle Maintain? (5)

'Canonical' Properties

Equivalent to LLVM's Loop Simplify Form

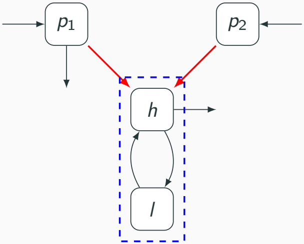
Every natural loop must have:

a **dedicated** preheader, **dedicated** exits, and a **unique** latch

Canonical Properties

Dedicated Preheader

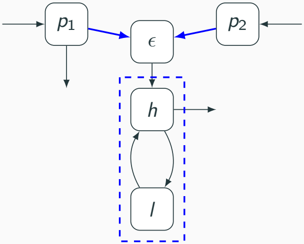
enables easy + efficient instruction hoisting



Canonical Properties

Dedicated Preheader

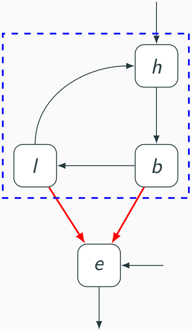
enables easy + efficient instruction hoisting



Canonical Properties

Dedicated Exit Blocks

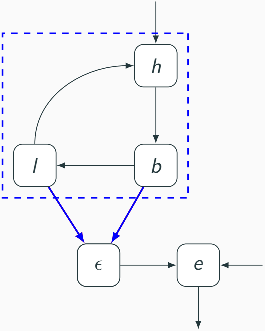
enables easy + efficient effect sinking



Canonical Properties

Dedicated Exit Blocks

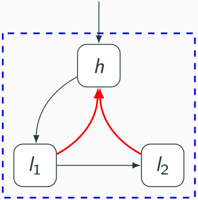
enables easy + efficient effect sinking



Canonical Properties

Unique Backedge + Latch

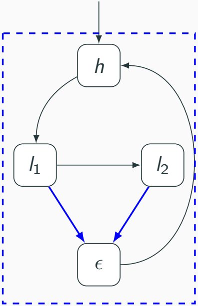
makes destruction of loop unambiguous



Canonical Properties

Unique Backedge + Latch

makes destruction of loop unambiguous



Graph Modifications

Operations

Observations

Operations

Observations

Edge can be deleted arbitrarily
Edge deletion affects a *bounded* subgraph

Operations

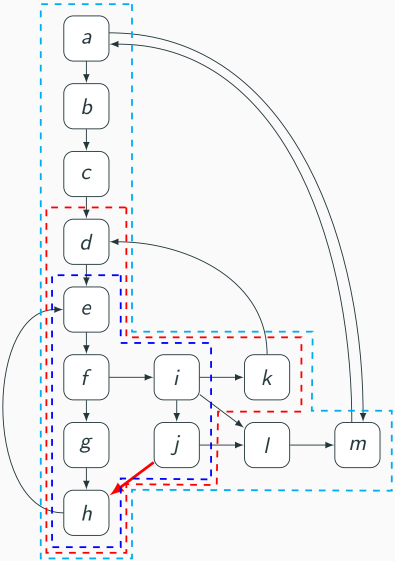
Observations

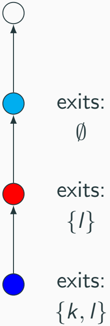
Edge can be deleted arbitrarily
Edge deletion affects a *bounded* subgraph

Edges **cannot** be added arbitrarily
Single-entry subgraphs can instead be *duplicated*
Preserves domination, loop structure, SSA and LCSSA properties

Edge Deletion: Simple Example

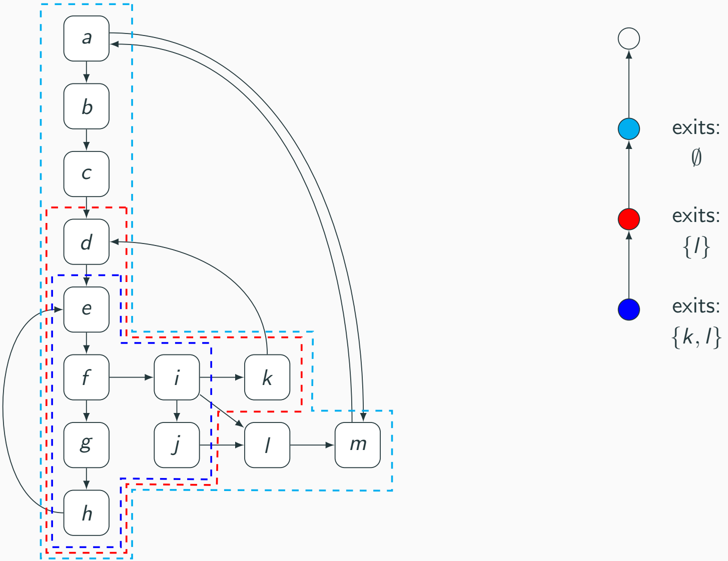
Edge Deletion: (j, h)





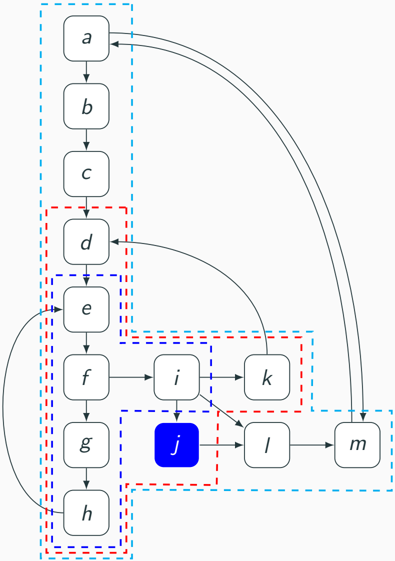
Initial graph

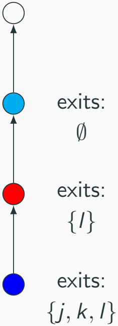
Edge Deletion: (j, h)



Edge deleted

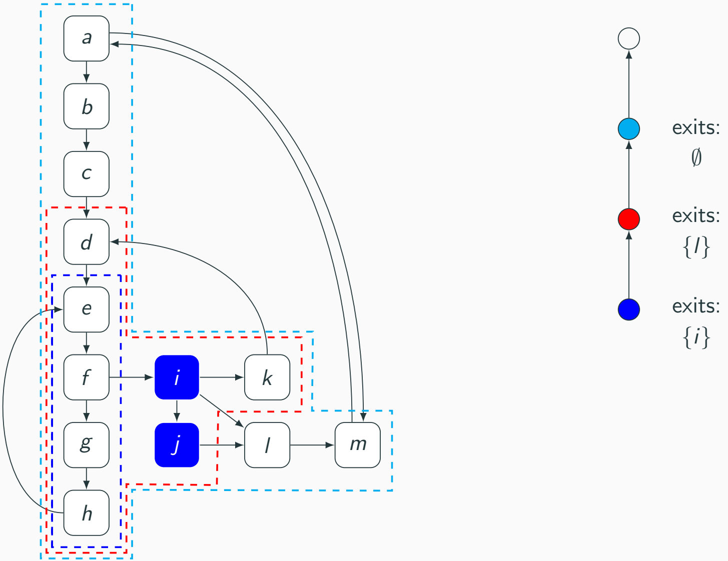
Edge Deletion: (j, h)





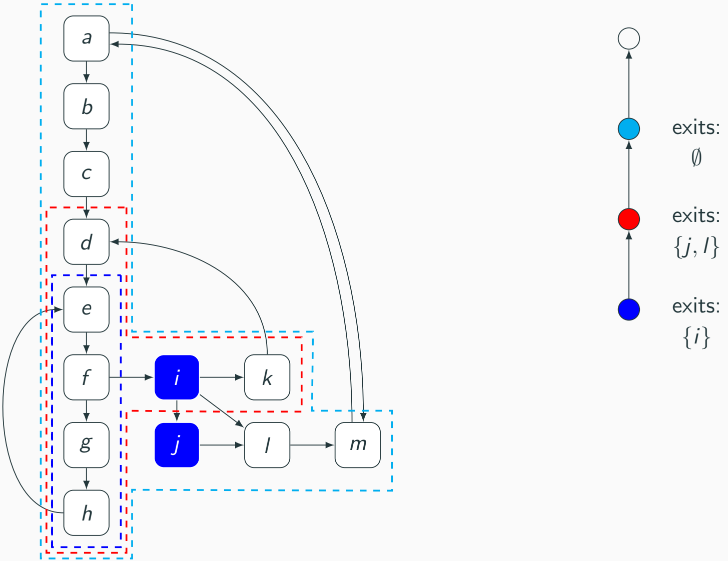
Eject block j from inner (blue) loop

Edge Deletion: (j, h)



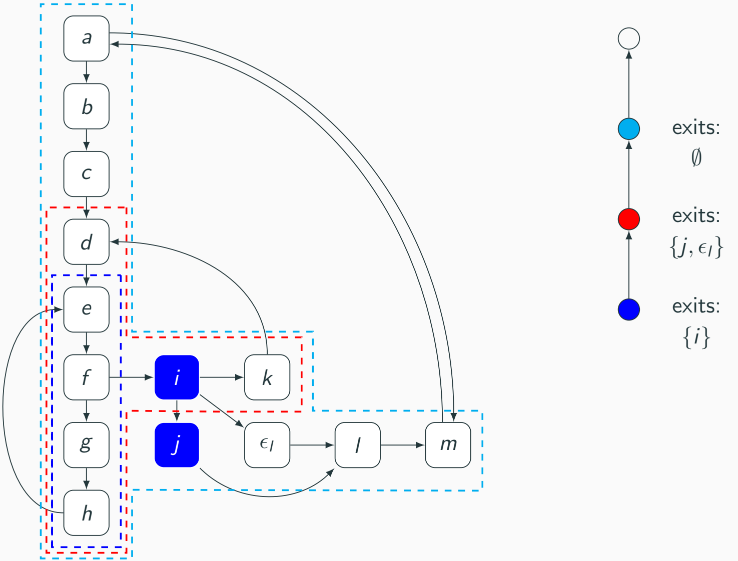
Eject block *i* from inner (blue) loop

Edge Deletion: (j, h)



Eject block j from middle (red) loop

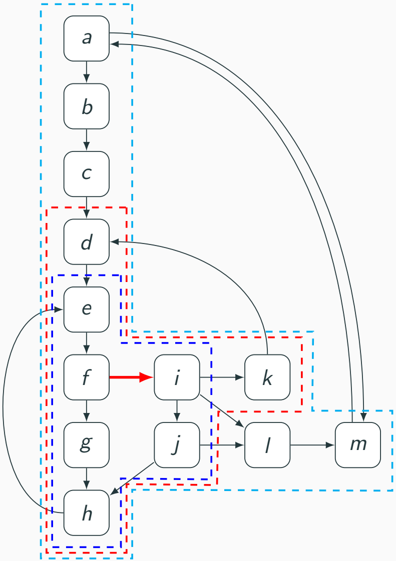
Edge Deletion: (j, h)

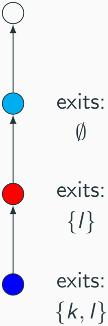


Place block ϵ_l on edge (i, l) to dedicate exit

Edge Deletion: Chaos Example

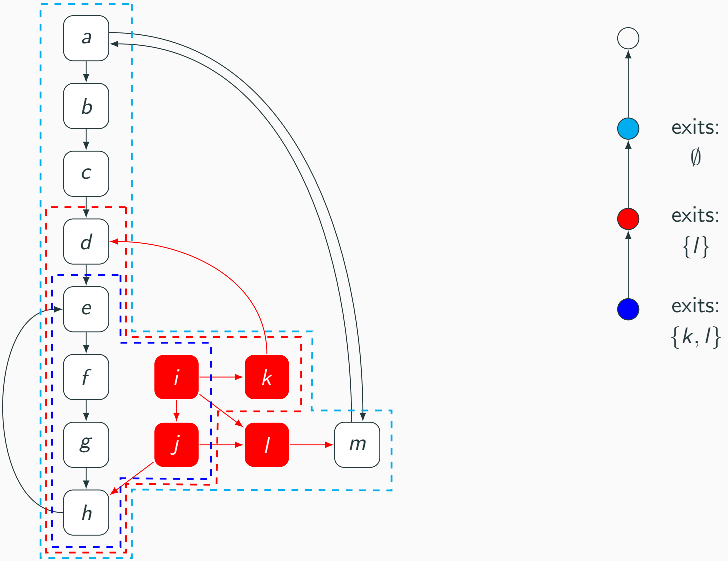
Edge Deletion: (f, i)





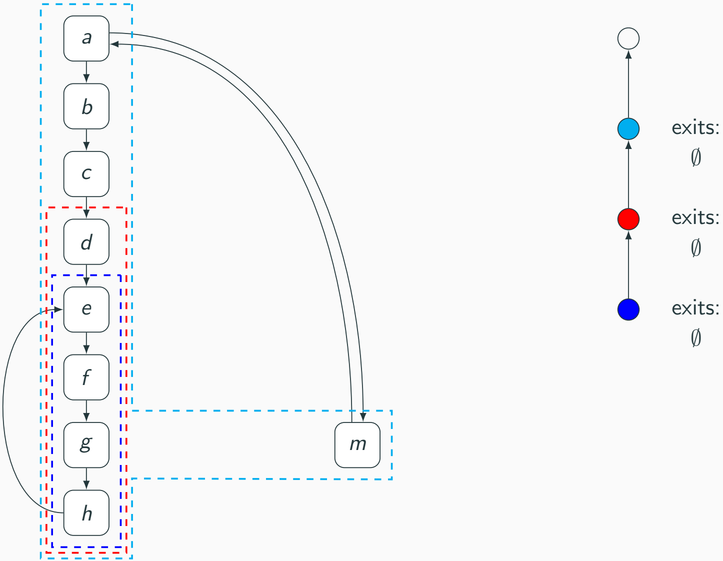
Initial graph

Edge Deletion: (f, i)



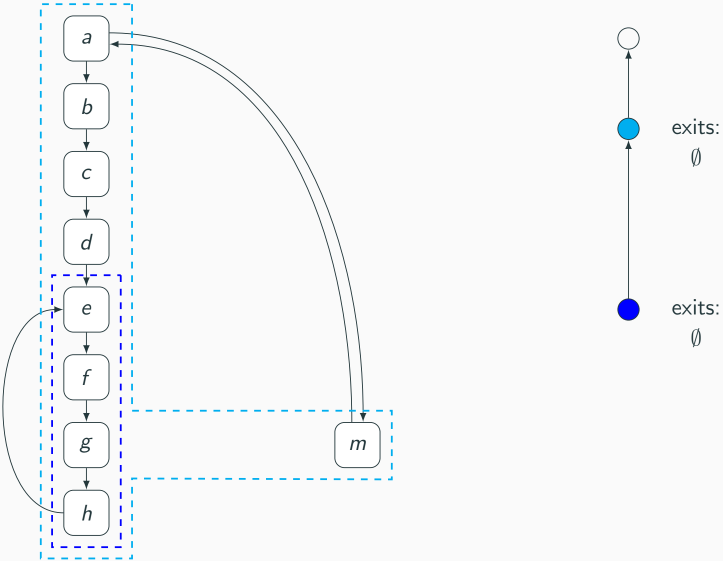
Edge deleted

Edge Deletion: (f, i)



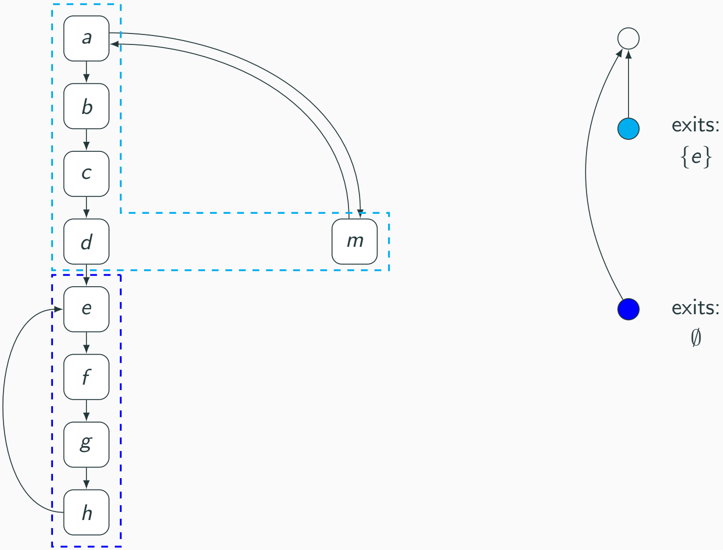
Remove unreachable blocks from graph, loop nesting forest

Edge Deletion: (f, i)



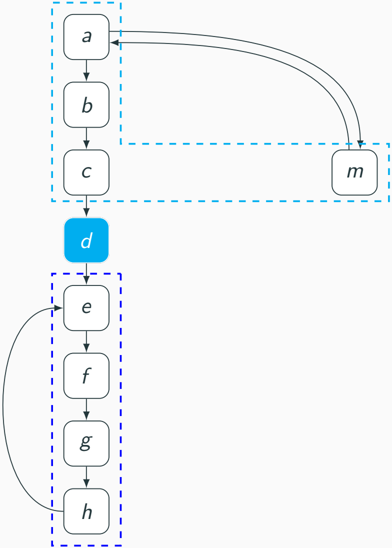
Remove destroyed middle (red) loop

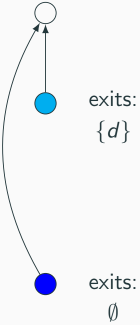
Edge Deletion: (f, i)



Eject block e (and its loop) from the outer (cyan) loop

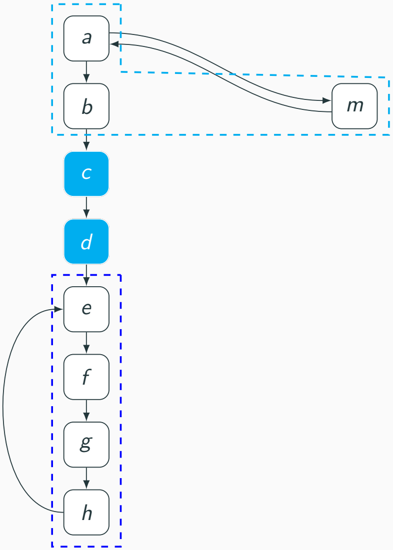
Edge Deletion: (f, i)

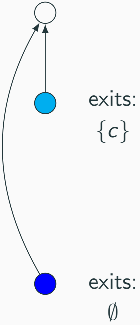




Eject block d from outer (cyan) loop

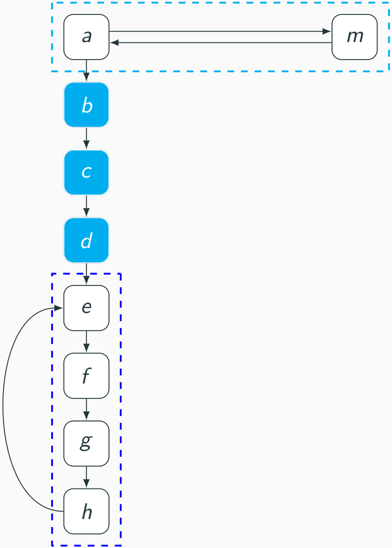
Edge Deletion: (f, i)

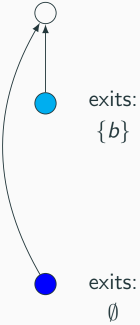




Eject block c from outer (cyan) loop

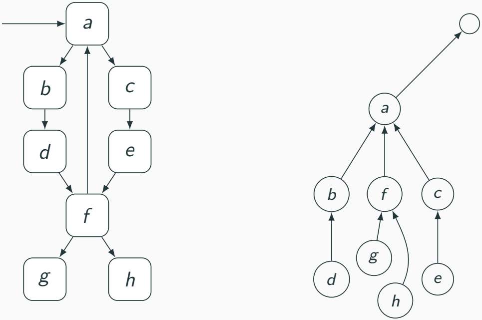
Edge Deletion: (f, i)



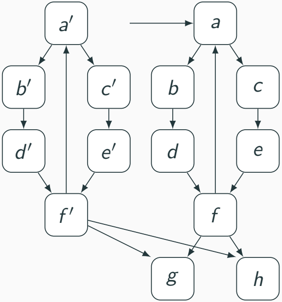


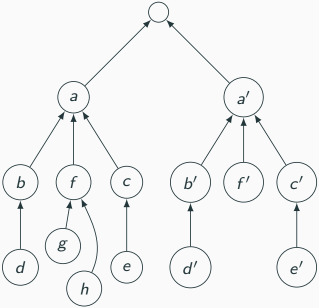
Eject block b from outer (cyan) loop

Subgraph Duplication (Dominator Tree)

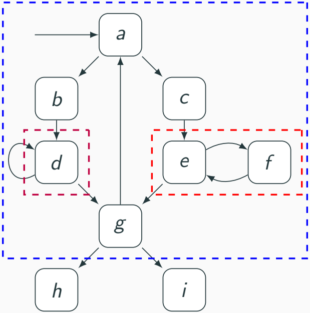


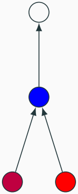
Subgraph Duplication (Dominator Tree)



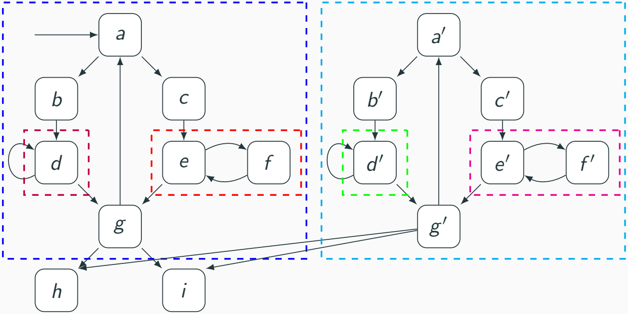


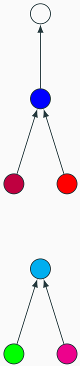
Subgraph Duplication (Loop Nesting Forest)





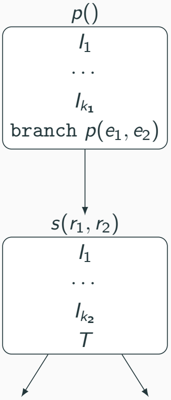
Subgraph Duplication (Loop Nesting Forest)





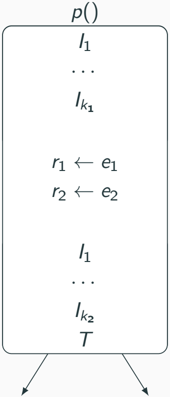
Straightening

Straightening (Example)



Find non-critical edge (where $pred(s) = \{p\} \land succ(p) = \{s\}$)

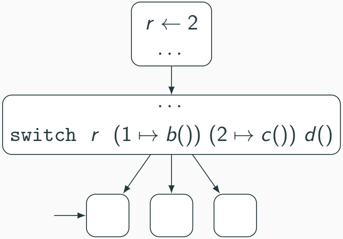
Straightening (Example)



Convert block parameters to move instructions

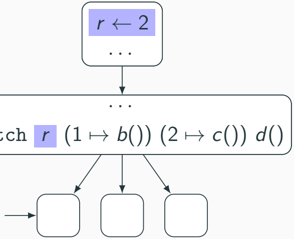
If Simplification

If Simplification (Example)



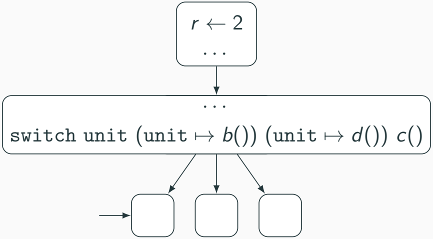
Initial graph

If Simplification (Example)



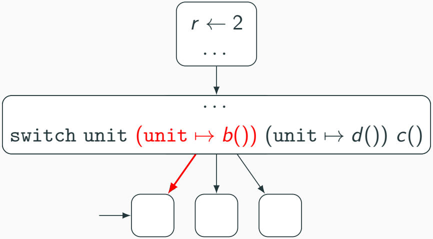
Switch target known statically

If Simplification (Example)



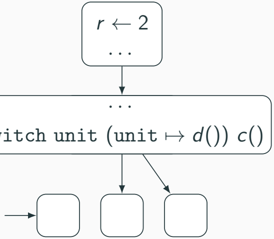
Rearrange terminator cases

If Simplification (Example)

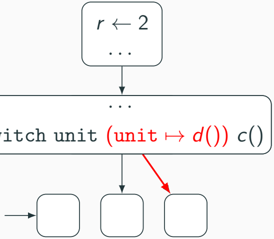


Run edge deletion on unit first case

If Simplification (Example)

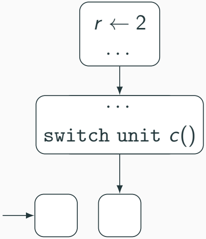


If Simplification (Example)



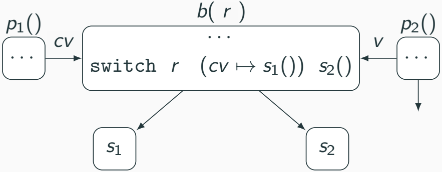
Run edge deletion on `unit` second case

If Simplification (Example)



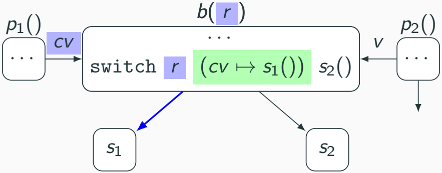
Jump Simplification

Jump Simplification (Example)



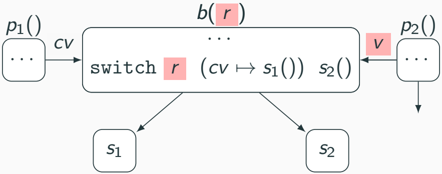
Initial graph

Jump Simplification (Example)



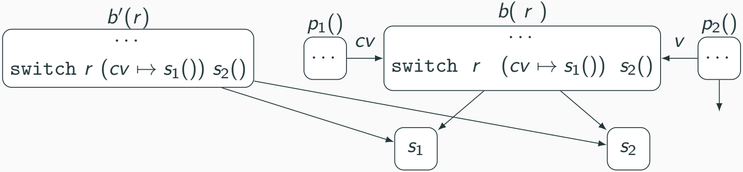
Switch target known statically on one path

Jump Simplification (Example)



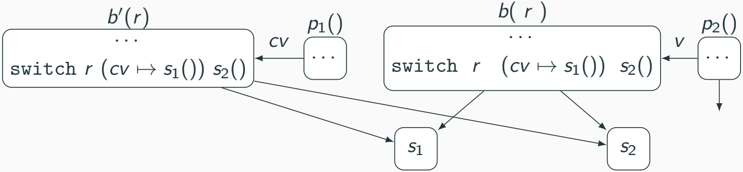
(Not necessarily all paths)

Jump Simplification (Example)



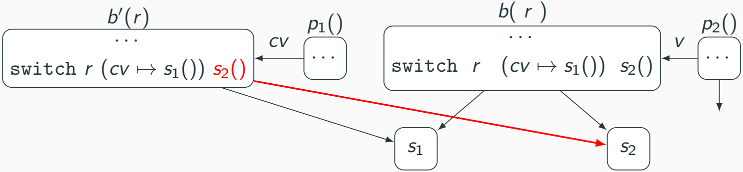
Duplicate block with switch

Jump Simplification (Example)



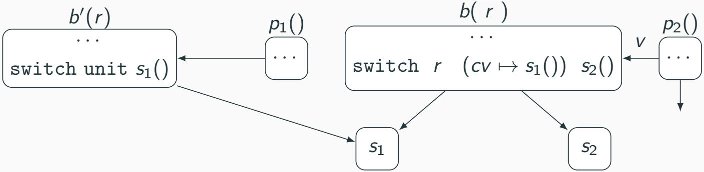
Thread the jump

Jump Simplification (Example)



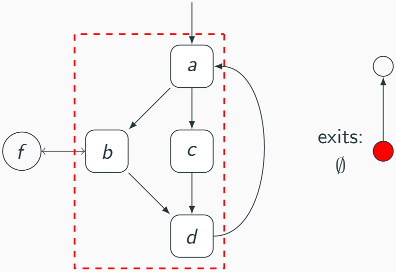
Run edge deletion on default case

Jump Simplification (Example)



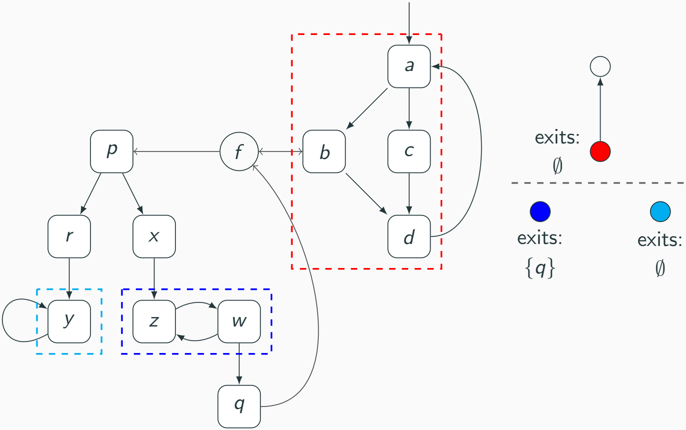
Function Inlining

Function Inlining (Example 1)



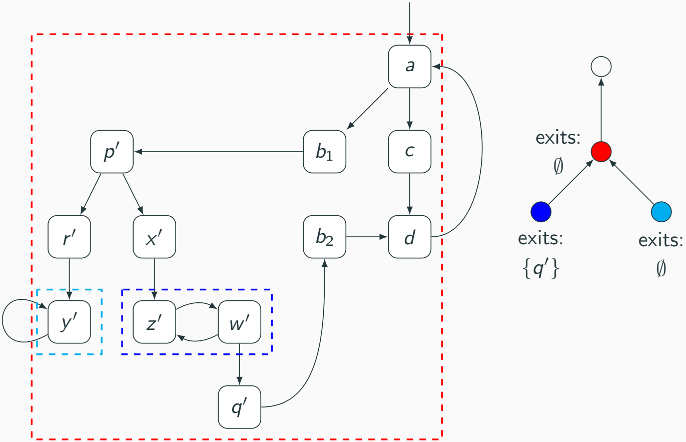
Initial graph

Function Inlining (Example 1)



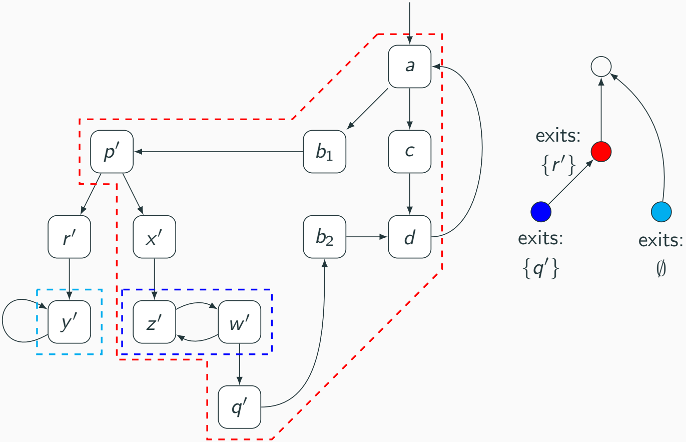
Initial graph with CFG/LNF of called function

Function Inlining (Example 1)



Inline call/return - merge loop structures

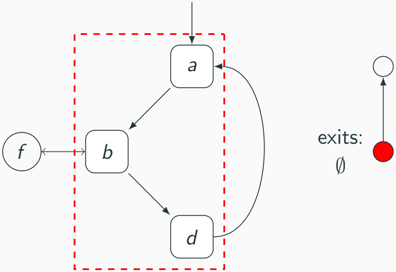
Function Inlining (Example 1)



Run block ejection on loop containing callsite

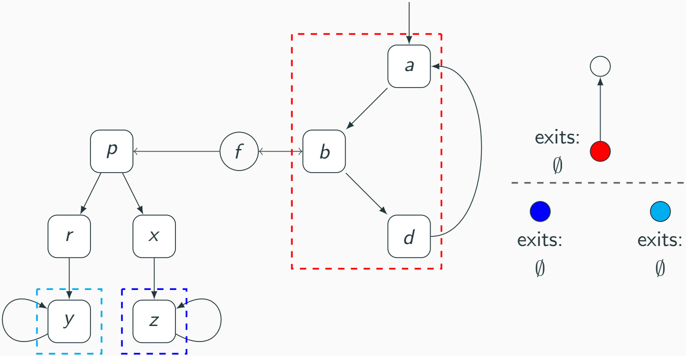
(Devil in the Details)

Function Inlining (Example 2)



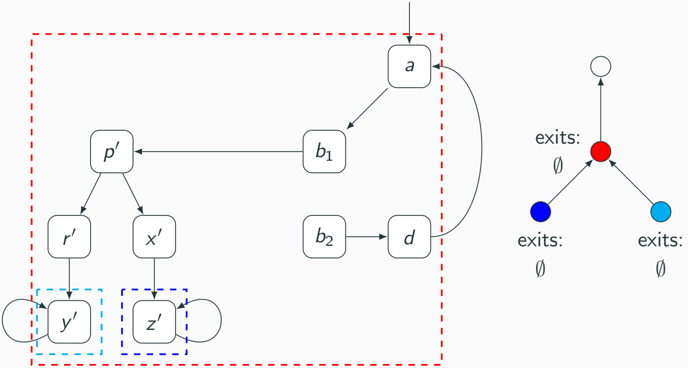
Initial graph

Function Inlining (Example 2)



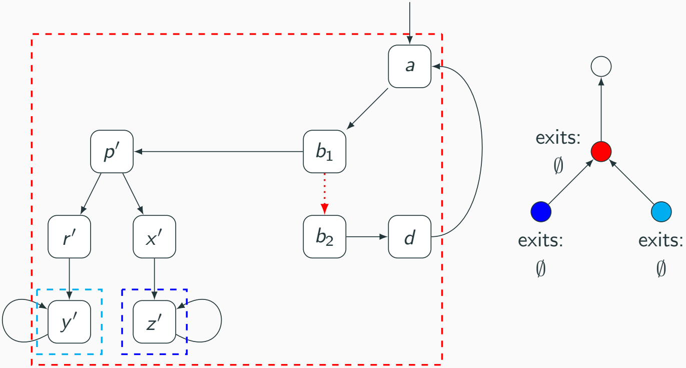
Initial graph with CFG/LNF of called function

Function Inlining (Example 2)



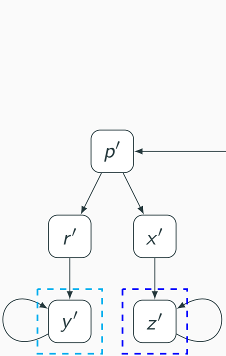
Inline call/return - merge loop structures

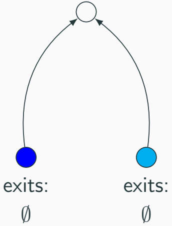
Function Inlining (Example 2)



Delete *fake* edge (b_1, b_2)

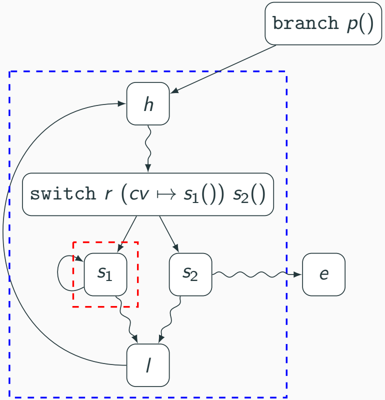
Function Inlining (Example 2)

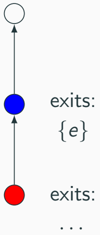




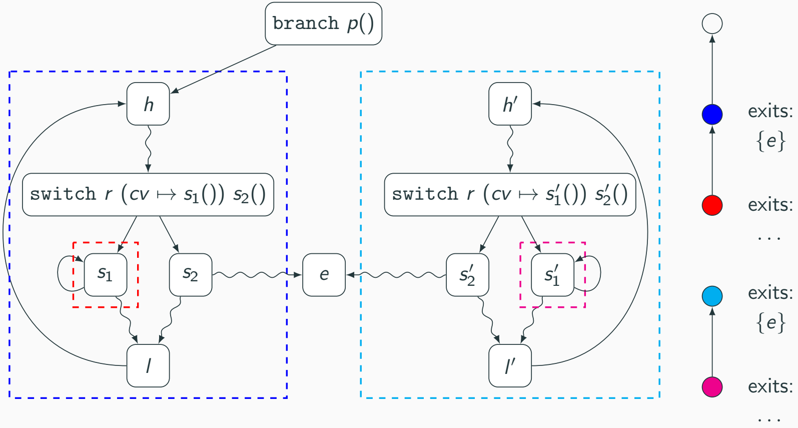
Loop Unswitching

Loop Unswitching (Example)



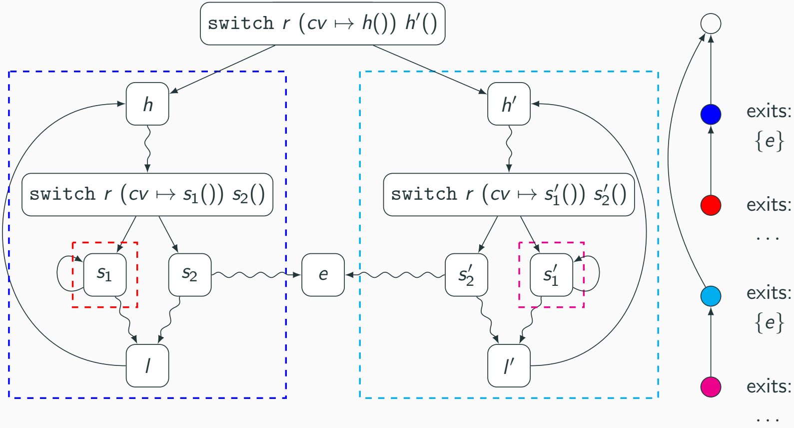


Initial graph



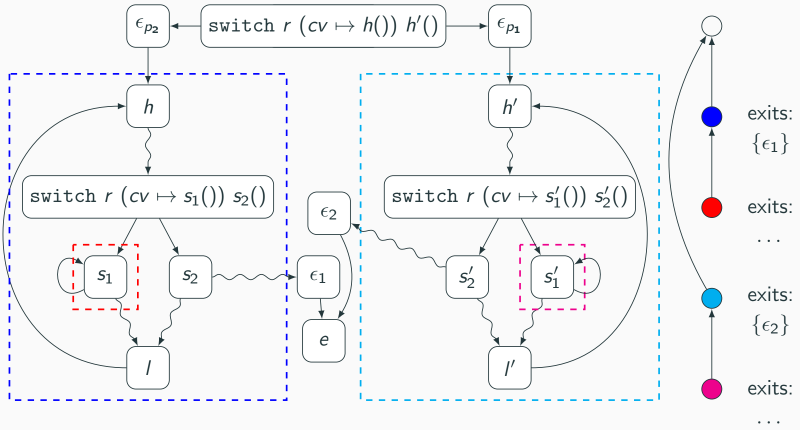
Clone loop containing switchable condition

Loop Unswitching (Example)



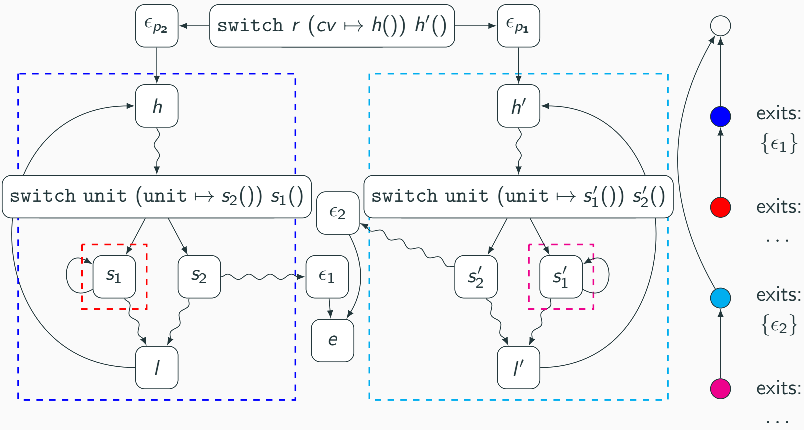
Update preheader to simulate switchable condition

Loop Unswitching (Example)



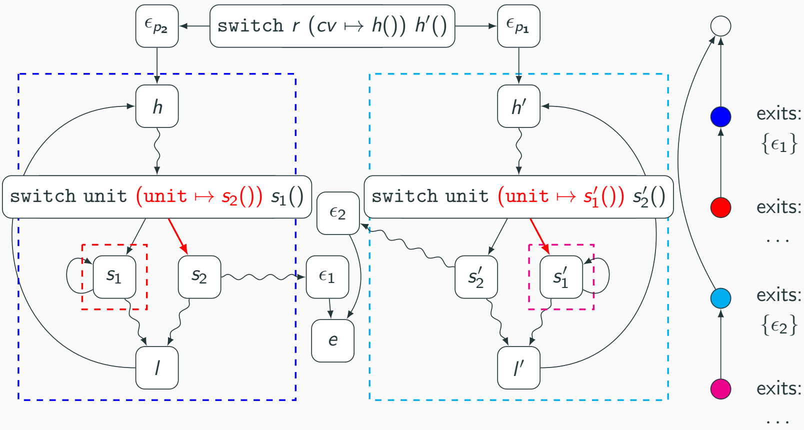
Dedicate preheader and exits

Loop Unswitching (Example)



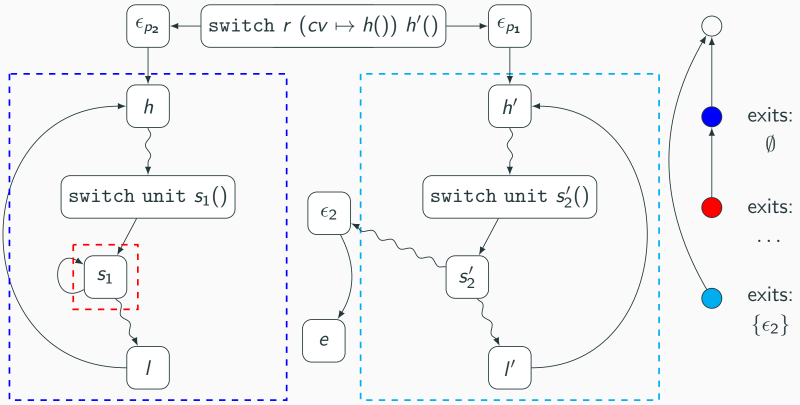
Rearrange terminator cases

Loop Unswitching (Example)



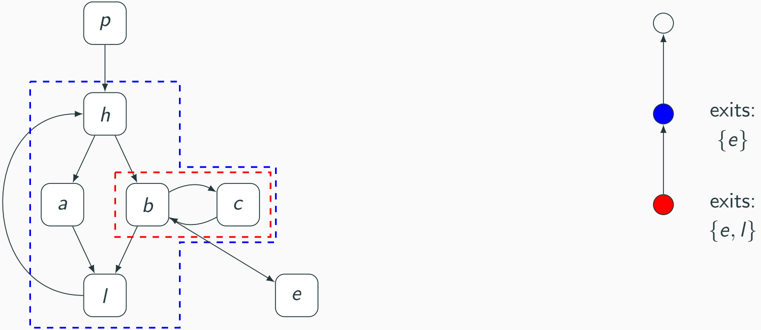
Run edge deletion on unswitched blocks

Loop Unswitching (Example)



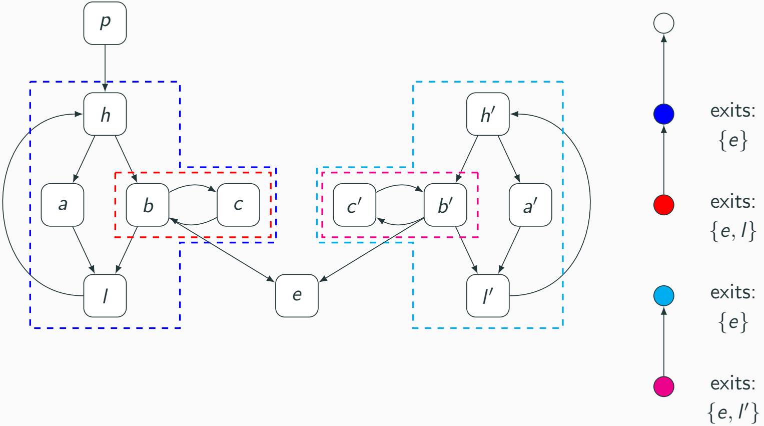
Loop Unrolling

Loop Unrolling (Example)



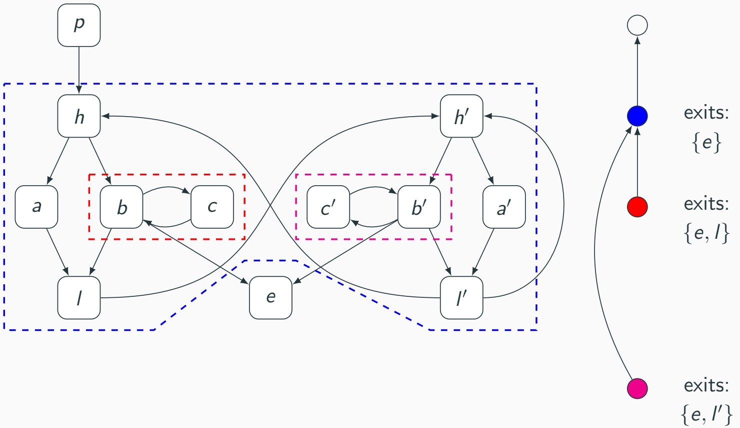
Initial graph

Loop Unrolling (Example)



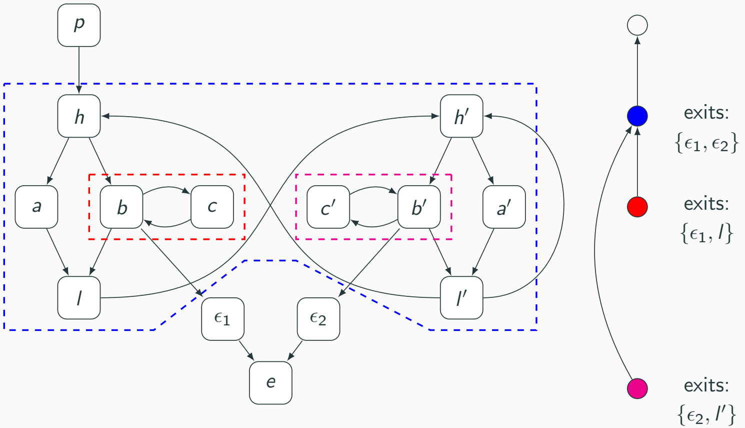
Duplicate loop

Loop Unrolling (Example)



Over, under, pull it tight ...

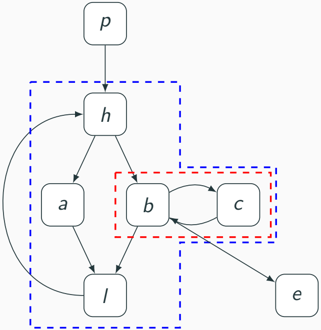
Loop Unrolling (Example)

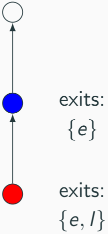


Dedicate exits

Loop Peeling

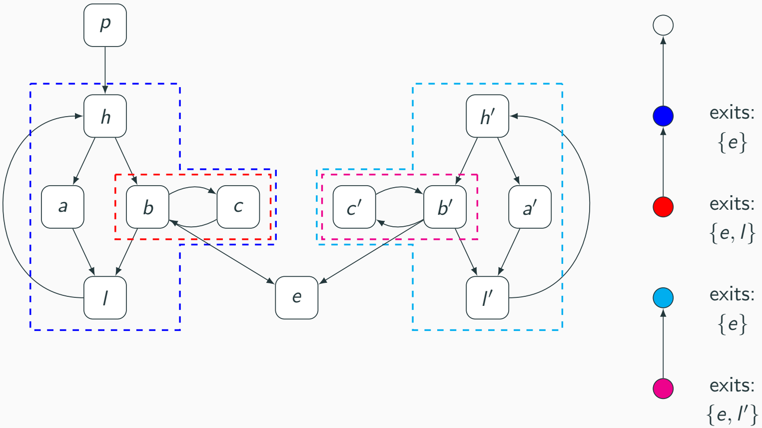
Loop Peeling (Example)





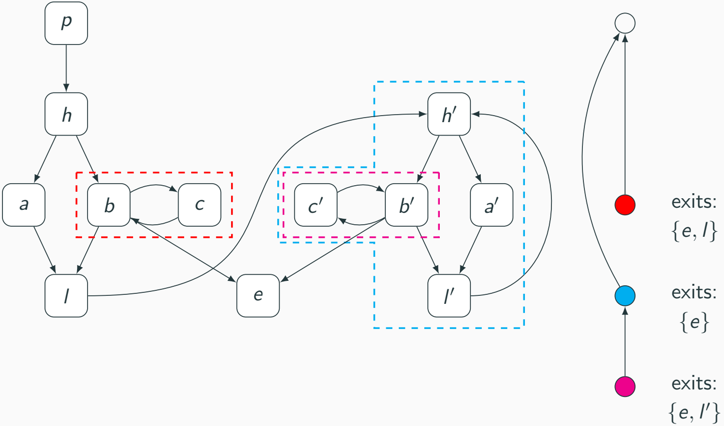
Initial graph

Loop Peeling (Example)



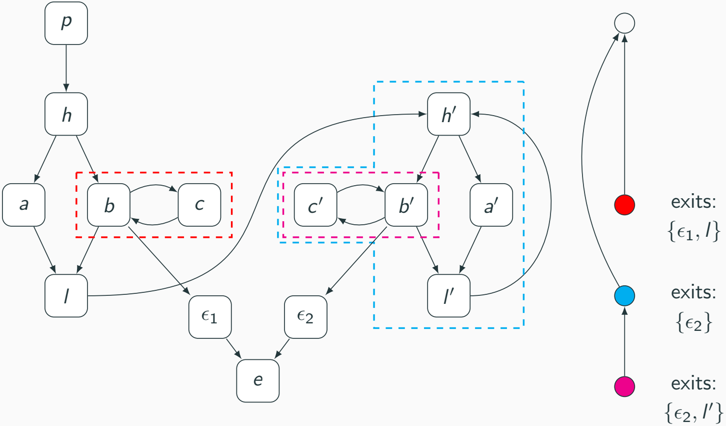
Duplicate loop

Loop Peeling (Example)



Usurp latch

Loop Peeling (Example)



Dedicate exits

Guarantees

Transformations

$$(f, D, H_F, L_F, X_F) \xrightarrow[args]{\mathsf{T}} (f_{out}, D_{out}, H_{out}, L_{out}, X_{out})$$

Transformations

$$\left(f, D, \underbrace{H_F, L_F, X_F}_{\textit{decomposition of loop nesting forest } F}\right) \xrightarrow[\textit{args}]{\mathsf{T}} \left(f_{out}, D_{out}, \overbrace{H_{out}, L_{out}, X_{out}}^{\textit{recomposes to loop nesting forest } F_{\textbf{out}}}\right)$$

Transformations

$$\underbrace{(f, D, \overbrace{}^{}H_F, L_F, X_F)}_{} $$

$$(f, D, \underbrace{H_F, L_F, X_F}_{\textit{decomposition of loop nesting forest } F}) \xrightarrow[args]{T} (f_{out}, D_{out}, \overbrace{H_{out}, L_{out}, X_{out}}^{\textit{recomposes to loop nesting forest } F_{\mathbf{out}}})$$

Note: $D \equiv D_f$ and $F \equiv F_f$ assusmed for all optimizations

Maintenance Properties

Theorem (Maintenance of Types)
If $p \mid f$ is well-typed and f is in SSA form, then $p[{}^{f}/_{f_{\mathbf{out}}}] \mid f_{out}$ is well-typed.

Maintenance Properties

Theorem (Maintenance of Types)
If $p \mid f$ is well-typed and f is in SSA form, then $p[{}^{f}/_{f_{\mathbf{out}}}] \mid f_{out}$ is well-typed.

Theorem (Maintenance of LCSSA Form)
If f is in LCSSA form, then f_{out} is in LCSSA form.

Maintenance Properties

Theorem (Maintenance of Types)
If $p \mid f$ is well-typed and f is in SSA form, then $p[{}^{f}/_{f_{out}}] \mid f_{out}$ is well-typed.

Theorem (Maintenance of LCSSA Form)
If f is in LCSSA form, then f_{out} is in LCSSA form.

Theorem (Maintenance of Canonical Form)
If f is in canonical form, then f_{out} is in canonical form.

Maintenance Properties

Theorem (Maintenance of Types)
If $p \mid f$ *is well-typed and* f *is in SSA form, then* $p[{}^{f}/_{f_{\mathbf{out}}}] \mid f_{out}$ *is well-typed.*

Theorem (Maintenance of LCSSA Form)
If f *is in LCSSA form, then* f_{out} *is in LCSSA form.*

Theorem (Maintenance of Canonical Form)
If f *is in canonical form, then* f_{out} *is in canonical form.*

Theorem (Maintenance of Dominator Tree)
The unique dominator tree of $G_{f_{\mathbf{out}}}$ *is* D_{out}*.*

Maintenance Properties

Theorem (Maintenance of Types)
If $p \mid f$ is well-typed and f is in SSA form, then $p[{}^{f}/_{f_{\mathbf{out}}}] \mid f_{out}$ is well-typed.

Theorem (Maintenance of LCSSA Form)
If f is in LCSSA form, then f_{out} is in LCSSA form.

Theorem (Maintenance of Canonical Form)
If f is in canonical form, then f_{out} is in canonical form.

Theorem (Maintenance of Dominator Tree)
The unique dominator tree of $G_{f_{\mathbf{out}}}$ is D_{out}.

Theorem (Maintenance of Loop Nesting Forest)
If f is in canonical form, then F_{out} reconstructed from $(H_{out}, L_{out}, X_{out})$ is the unique loop nesting forest of $G_{f_{\mathbf{out}}}$.

IR Semantic

Small-step Reduction

$$(\langle p, f, b\rangle \mid \gamma \mid \nu \mid \mu \mid \Psi;\ s) \rightarrow (\langle p, f', b'\rangle \mid \gamma' \mid \nu' \mid \mu' \mid \Psi';\ s')$$

Streams

$$s = I_1, \ldots, I_k, T, \hat{s} \qquad\qquad \hat{s} = \langle f, b, r, s\rangle \mid \epsilon$$

Contexts

(registers)	$\gamma\ : R \rightarrow cv$
(memory)	$\mu\ : \mathbb{N} \rightarrow \{0, 1\}$
(effects)	$\Psi = \langle\overline{\psi}\rangle$
	$\psi = \hat{f}(\overline{v_i}) \mid \texttt{halt}(v) \mid \texttt{halt}(\textbf{ex}(\text{err}))$
(nondeterminism)	ν

Theorem (Semantic Equivalence)
Let $p' = f[^{f}/_{f_{\mathbf{out}}}]$ and let $\sigma_{ref} = [^{ref\ f}/_{ref\ f_{\mathbf{out}}}]$. If there exists an n-step evaluation of f such that

$$(p \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f(\overline{cv_{t_i}})) \rightarrow_{\rho}^{n} (\langle p, f_{t_1}, b_{t_1} \rangle \mid \gamma_1 \mid \mu' \mid \nu' \mid \Psi';\ s_{t_1})$$

then there exists a symmetric n'-step evaluation of f_{out} such that

$$(p' \mid \gamma \mid \mu \mid \nu \mid \Psi;\ f_{out}(\overline{cv_{t_i}[\sigma_{ref}]}) \rightarrow^{n'} (\langle p', f_{t_2}, b_{t_2} \rangle \mid \gamma_2 \mid \mu' \mid \nu' \mid \Psi'[\sigma_{ref}];\ s_{t_2})$$

and vice versa.

Evaluation

Methodology

Baseline:

Canonicalize Program

Build worklist of optimizations (for a particular optimization)

Perform optimizations without maintaining properties

Rebuild canonical form at end

Methodology

Baseline:

Canonicalize Program

Build worklist of optimizations (for a particular optimization)

Perform optimizations without maintaining properties

Rebuild canonical form at end

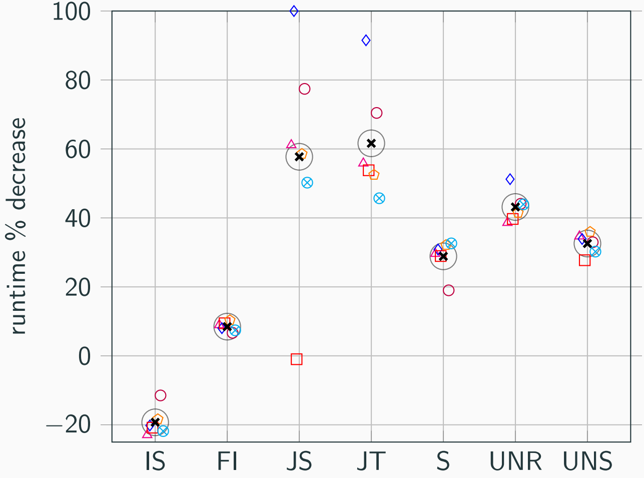
Comparison:

Canonicalize Program

Build worklist of optimizations (for a particular optimization)

Perform optimizations while maintaining properties

Evaluation Results



To Summarize

Contributions

- Description of Incremental Optimizer Construction Methodology
- Formalized Kernel IR (with deterministic semantics)
- Proof-of-Concept Implementation
- *Correctness* Evaluation (maintenance proofs)
- Runtime Evaluation

Let’s Discuss!