GRAPH IRs FOR IMPURE HIGHER-ORDER LANGUAGES

Oliver Bračevac

Guannan Wei

Songlin Jia

Supun Abeysinghe

Luke Jiang

Yuyan Bao

Tiark Rompf

|galois|

PURDUE UNIVERSITY

PURPL

AUGUSTA UNIVERSITY

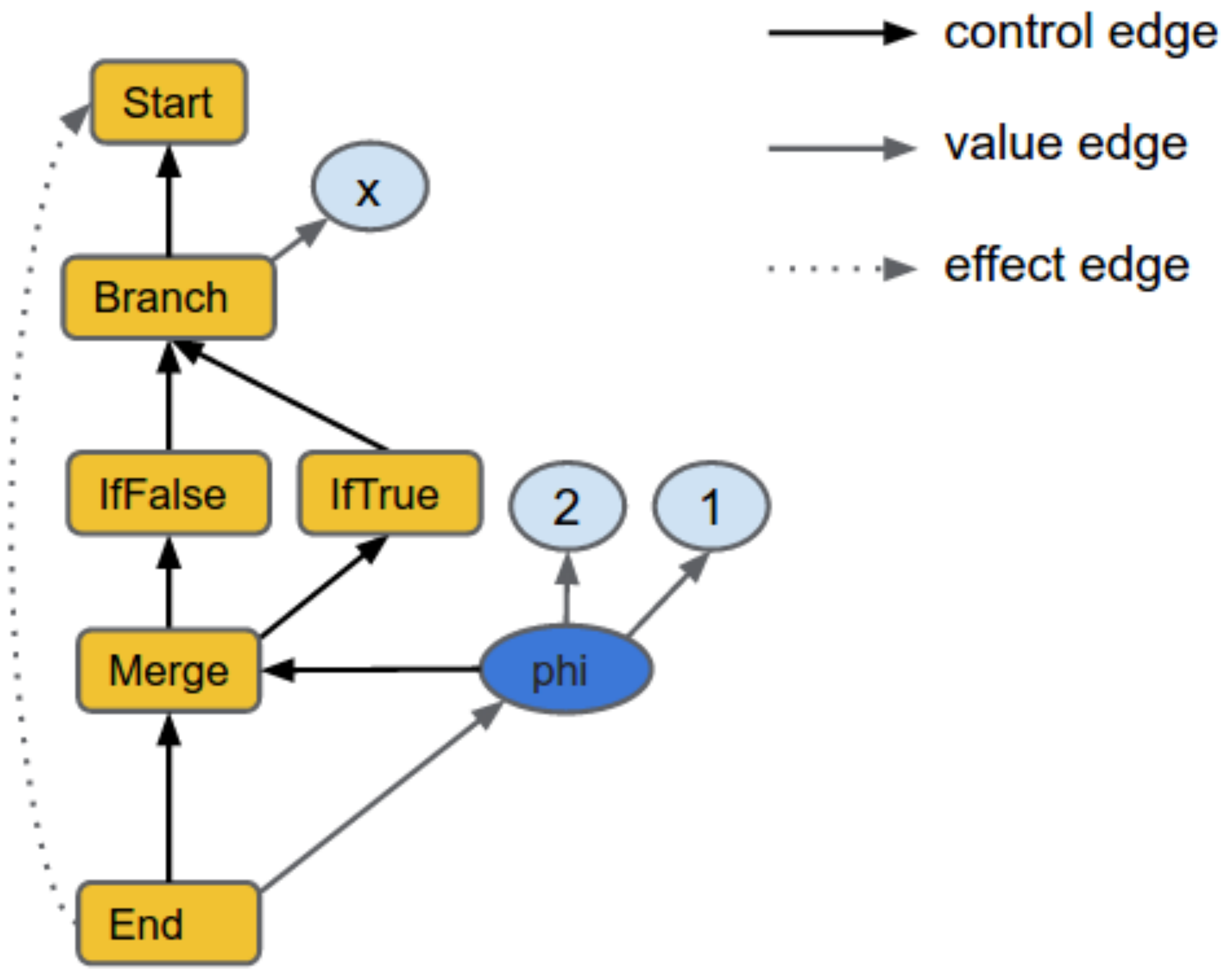
SEA-OF-NODES IRs

SEA-OF-NODES & GRAPH IRs

- **Imperative languages**: used in optimizing compilers & runtimes for (Java & JavaScript).
- **Pure functional languages:** used for graph reduction/term-graph rewriting (not considered here).
- **Sea-of-nodes:** dissolve programs into **graphs with data and control edges.**
  - Relaxed execution order & **highly localized** reasoning at nodes through dependencies.
  - More flexible & aggressive optimizations, code motion [Click 1995].
- Limitations
  - **Intraprocedural/local optimization scope!**
  - **First-order languages!**

Now what?

```
function (x) { return f(x) ? 1 : 2; }
```



Source: V8 TurboFan https://v8.dev/blog/turbofan-jit

SEA-OF-NODES & GRAPH IRs

How Could They Support Higher-Order Languages with Effects?

Say, what is the graph for this program?

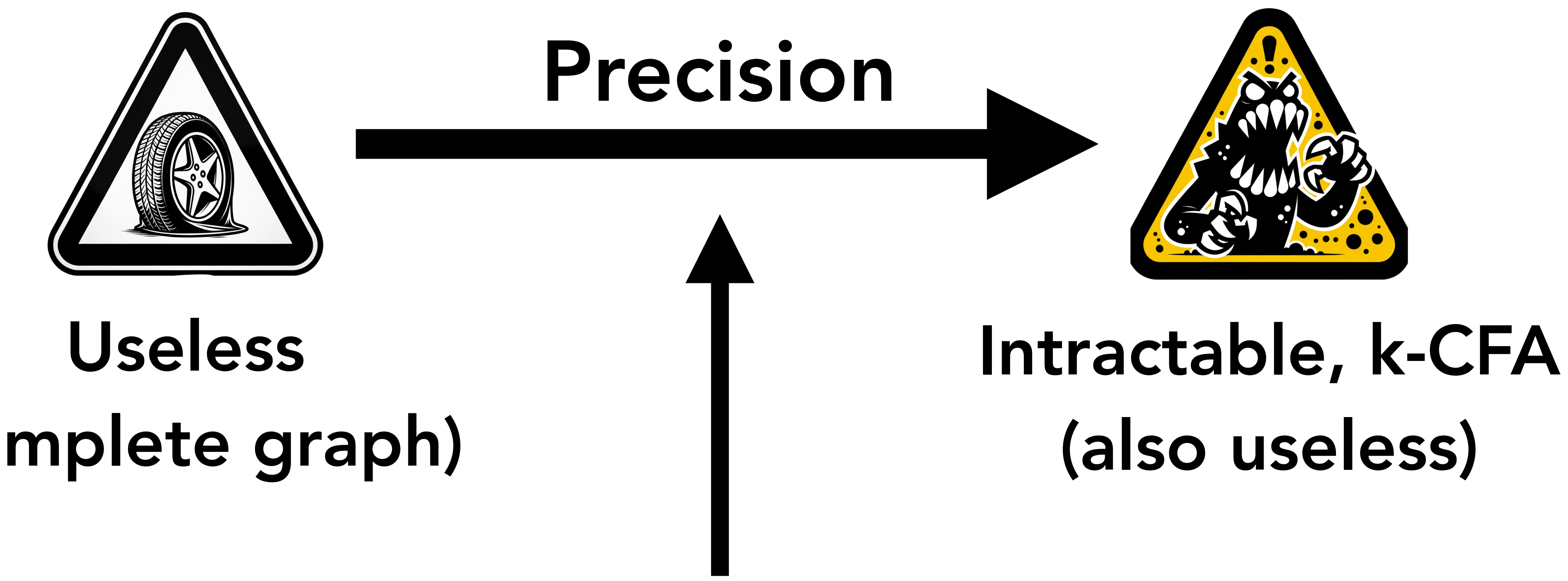
```
def map(f: Int => Int) = List(1, 2, 3).map(f)

val c = new Ref(0)

map(i => c := c+1; !c)

val d = c

val res = !d

map(i => if (i == 0) free(c); i)

res
```

Imprecise Control Transfers ⚠

Aliasing ⚠

Effects ⚠

How do we obtain dependencies?
Precision vs. Cost



How do we achieve a *reasonable* price-to-performance ratio?

Is this necessarily whole program?

IMPLEMENTED IN SCALA LMS!

THIS WORK

EFFECTS

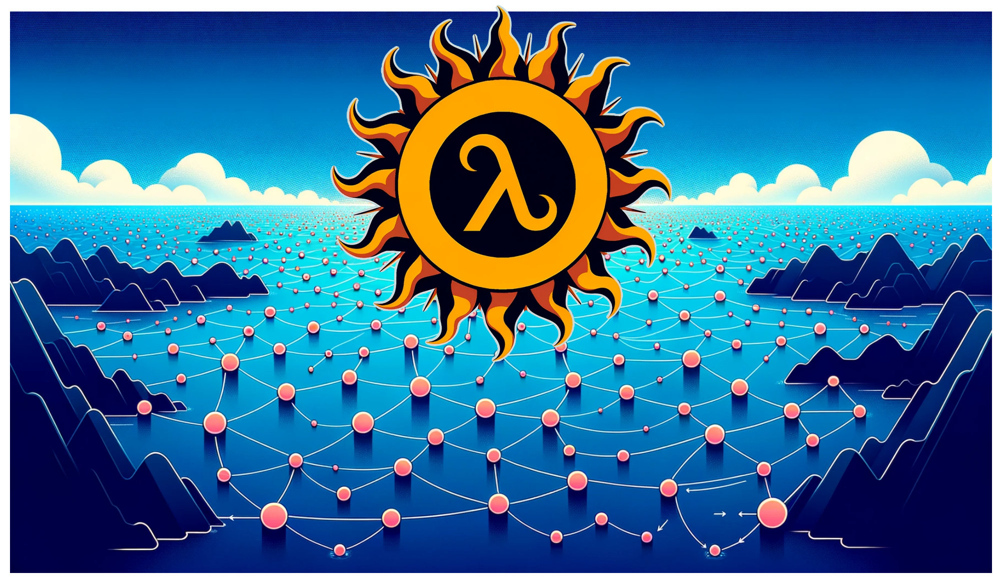
SEA-OF-NODES IRs

GLOBAL

MODULAR

AFFORDABLE

APPROACH IN A NUTSHELL



Problem: *Modular, Affordable, and Precise Dependencies*
for Higher-Order Programs + Effects

Solution:

- ***(Path-)Dependent types*** carry all relevant information (e.g., ***context and capabilities***).
- ***Type inference*** which is ***efficient in practice*** replaces expensive flow analyses.
  - ***General-purpose languages***: rely on ***lightweight user annotations*** (no worse than Rust).
  - ***DSLs***: types built into language constructs, ***no user annotation needed***.
- Naturally inherits the virtues of type systems: ***separate compilation***.

Foundation: *Reachability Types*
Seamless Ownership Types for Higher-Order Programs + Effects
(OOPSLA'21, Conditionally accepted at POPL'24)

REACHABILITY TYPES

OOPSLA'21, Successor Conditionally Accepted at POPL'24

```
val c1 : Ref[Int]^{c1}

val c2 : Ref[Int]^{c2}

val c3 : Ref[Int]^{c1,c3} = c1
```

Reachability sets track aliasing*

addRef's implementation ***must not share aliasing*** with its argument: $\varnothing \sqcap \{c_1\} = \varnothing$

addRef's implementation ***reaches/closes over*** c1.

```
def addRef(x : Ref[Int]^{∅}) =     // ((x: Ref[Int]^{∅}) ⇒ Ref[Int]^{c1} {c1,x})^{c1}
  c1 := !c1 + !x; c1
```

Dependent effect!

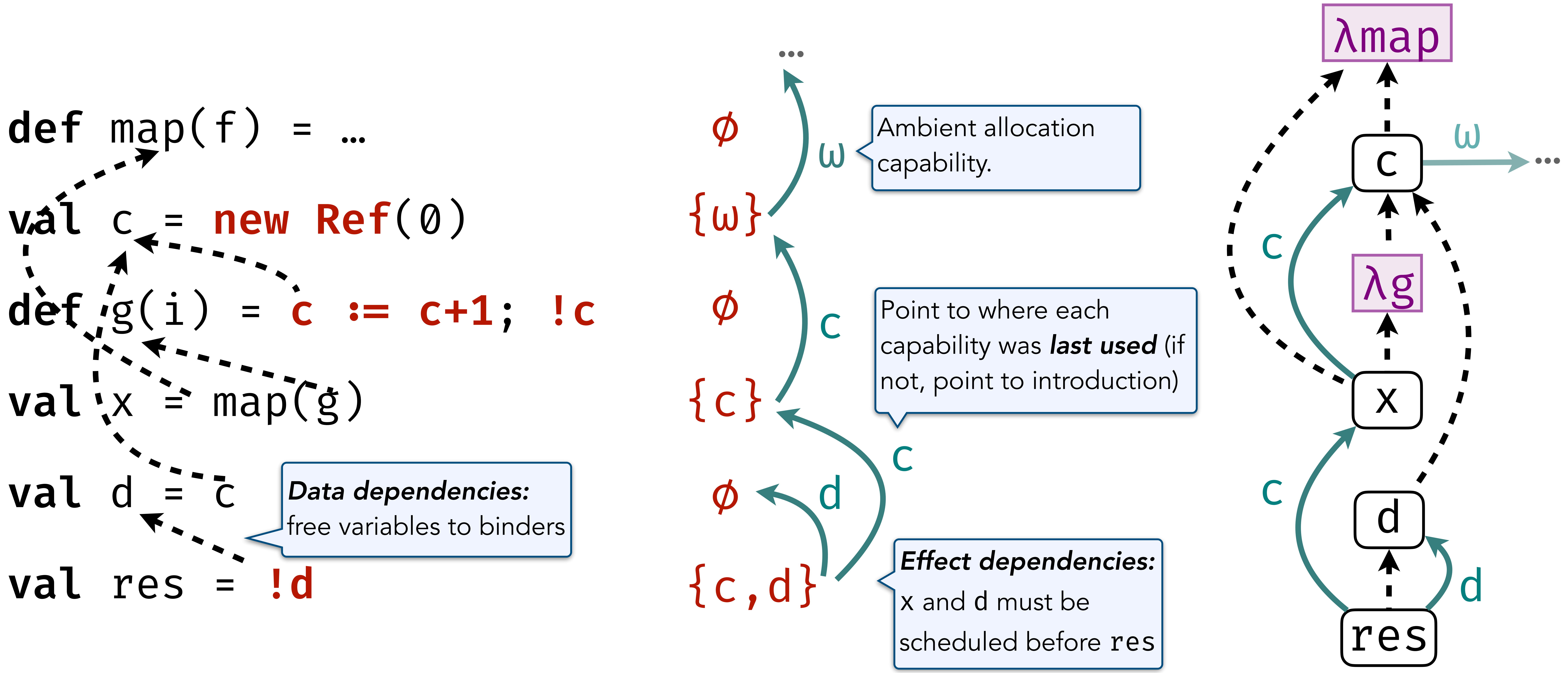
```
addRef(c2) // ok: separate      // ((x: Ref[Int]^{∅}) ⇒ Ref[Int]^{c1} w:{c1} r:{c1,x})^{c1}

addRef(c1) // error: overlap

addRef(c3) // error: overlap
```

Refinement: reads and writes

*it is actually a stronger relation! We *do not* need full alias analysis

TYPES AND EFFECTS → GRAPHS

ANF Informs Nodes and Data Edges, Effects Inform Control Edges



```
def map(f) = …
val c = new Ref(0)
def g(i) = c := c+1; !c
val x = map(g)
val d = c
val res = !d
```

Data dependencies: free variables to binders

Ambient allocation capability.

Point to where each capability was ***last used*** (if not, point to introduction)

Effect dependencies: `x` and `d` must be scheduled before `res`

LEXICAL STRUCTURE

Lambda, the Ultimate Graph IR!

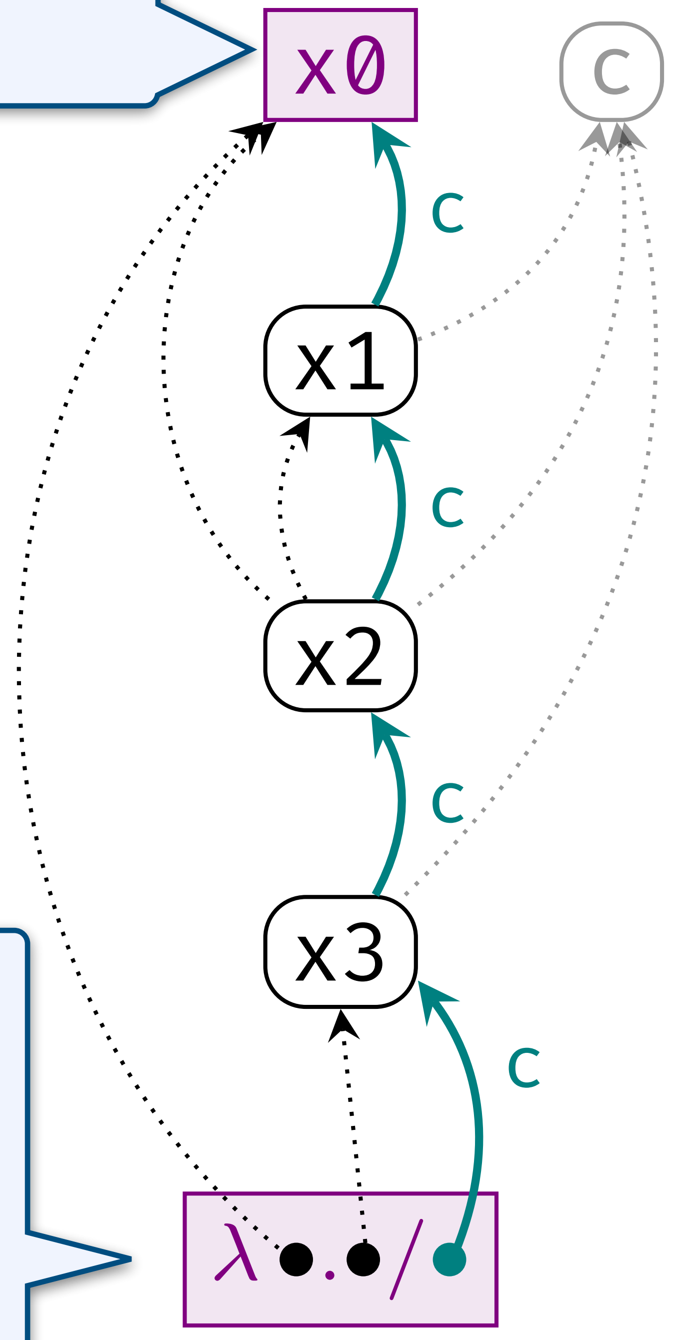
```
val c = new Ref(0)

def inc(x0: Int) = {
  val x1 = !c
  val x2 = c := x1 + x0
  val x3 = !c
  x3
}
```

Block-start variable:

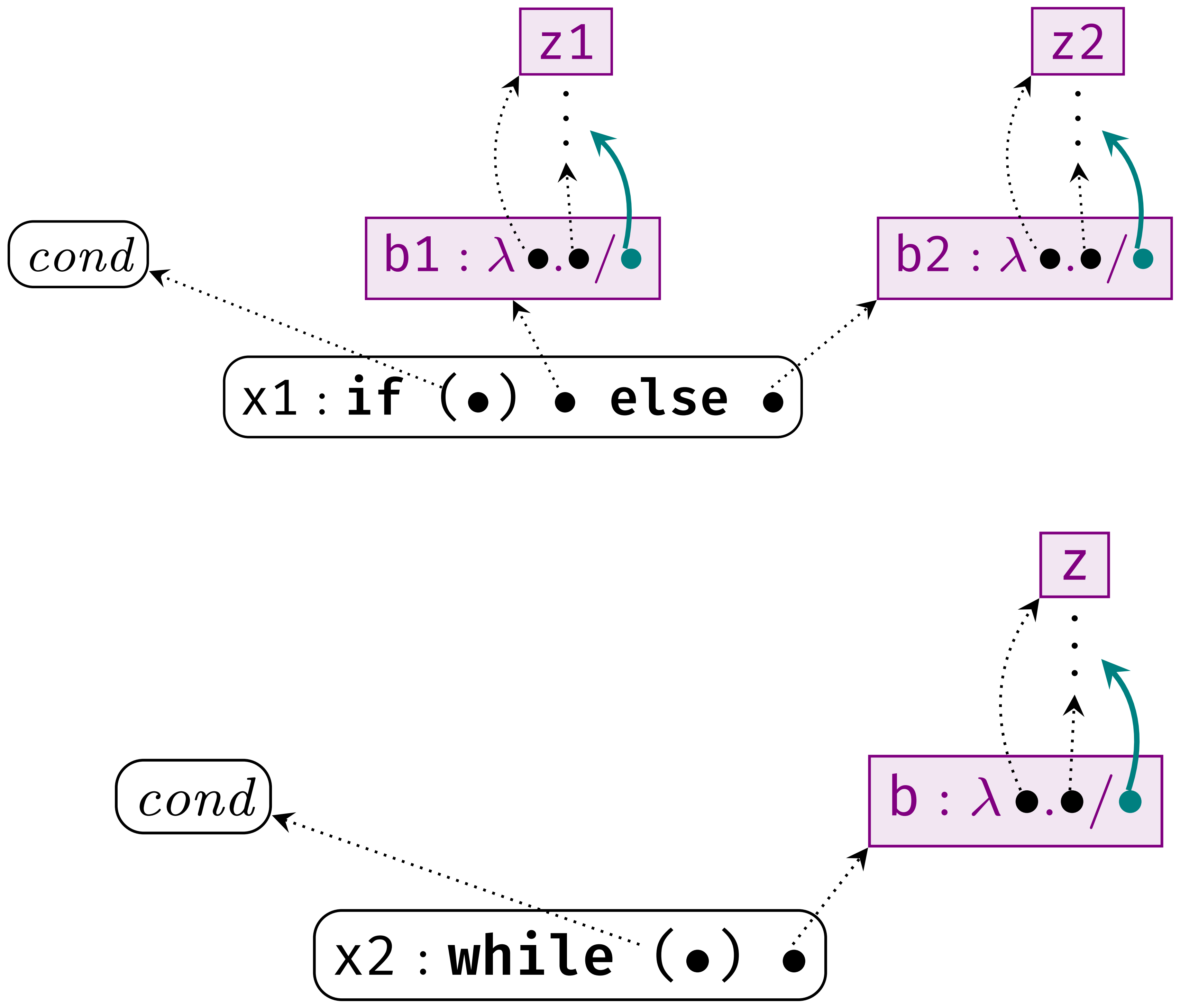
- Argument placeholder
- Control-flow predecessor placeholder

Lambda node:

- Block-start variable
- (+Self-variable if recursive)
- Return node
- Effect summary (for control-flow successors)



Control Structures from Lambda Blocks:

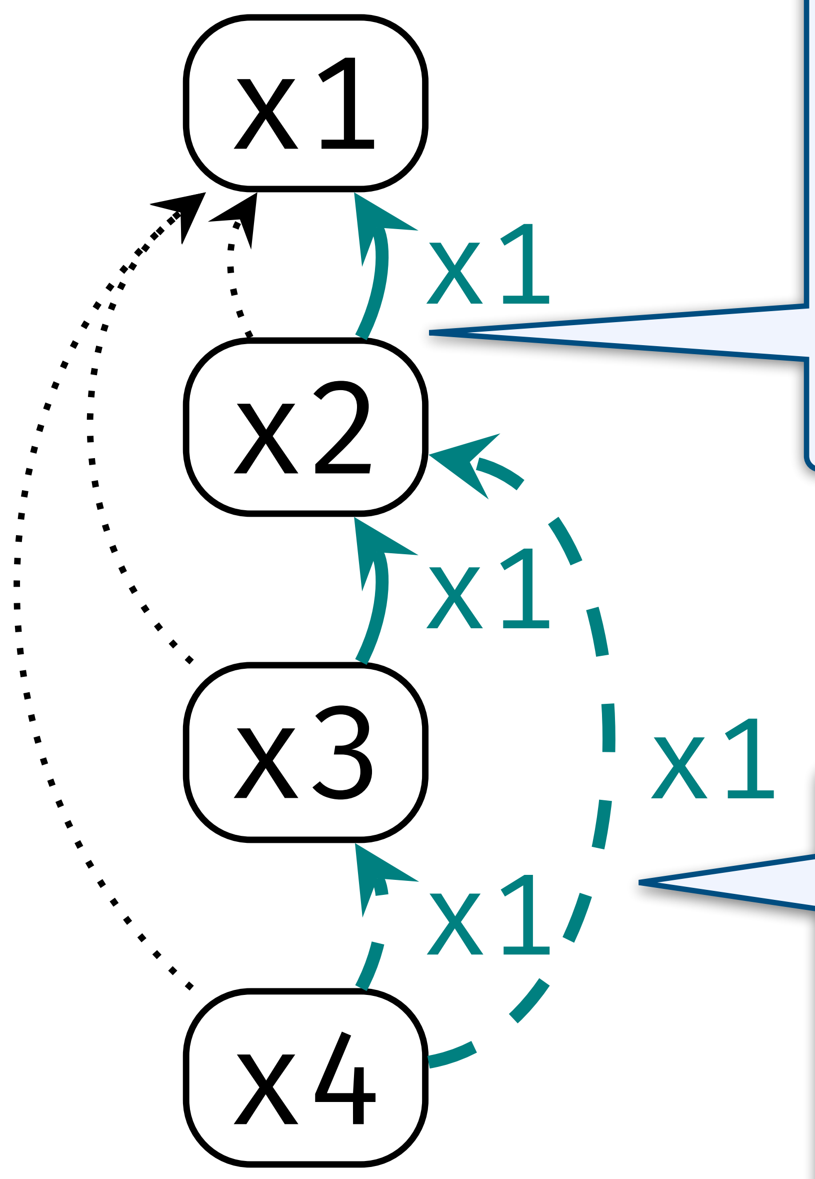


ANTI DEPENDENCE

From Read vs. Write Distinction

```
val x1 = new Ref(0)
~~val x2 = x1 := 21~~
~~val x3 = !x1~~
val x4 = x1 := 42
```

Needed?

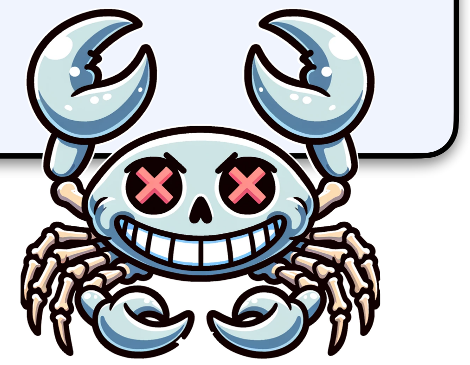


Hard dependency: strict adherence (as before).
If the current node is scheduled, then the target must have been scheduled beforehand.

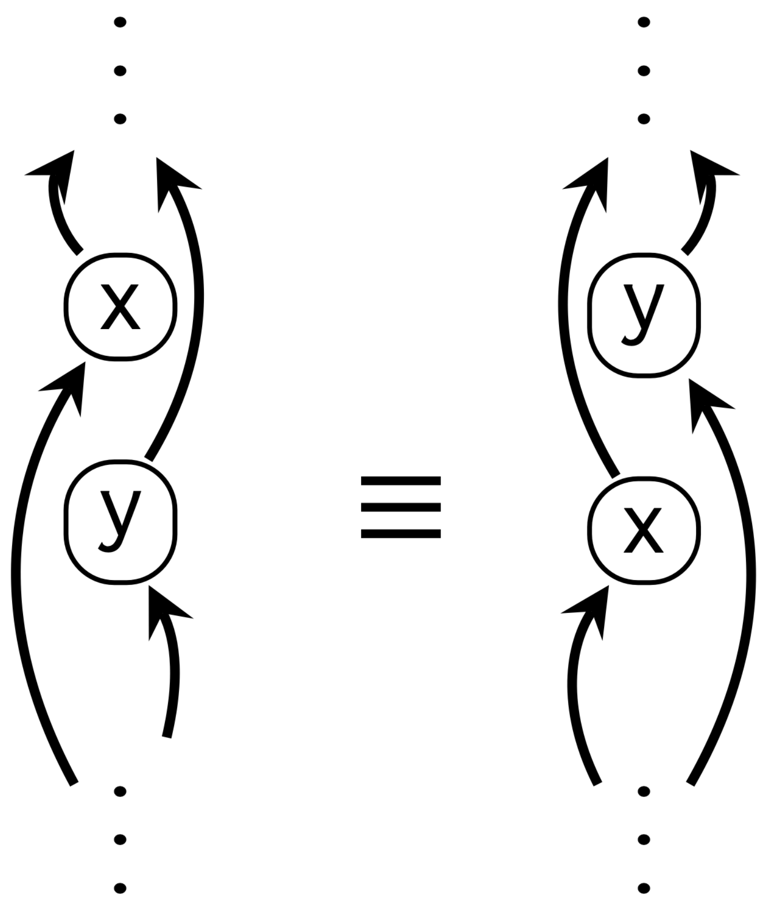
Soft dependency: current node may be scheduled even if the target isn't.

More Effect Goodness in the Paper!

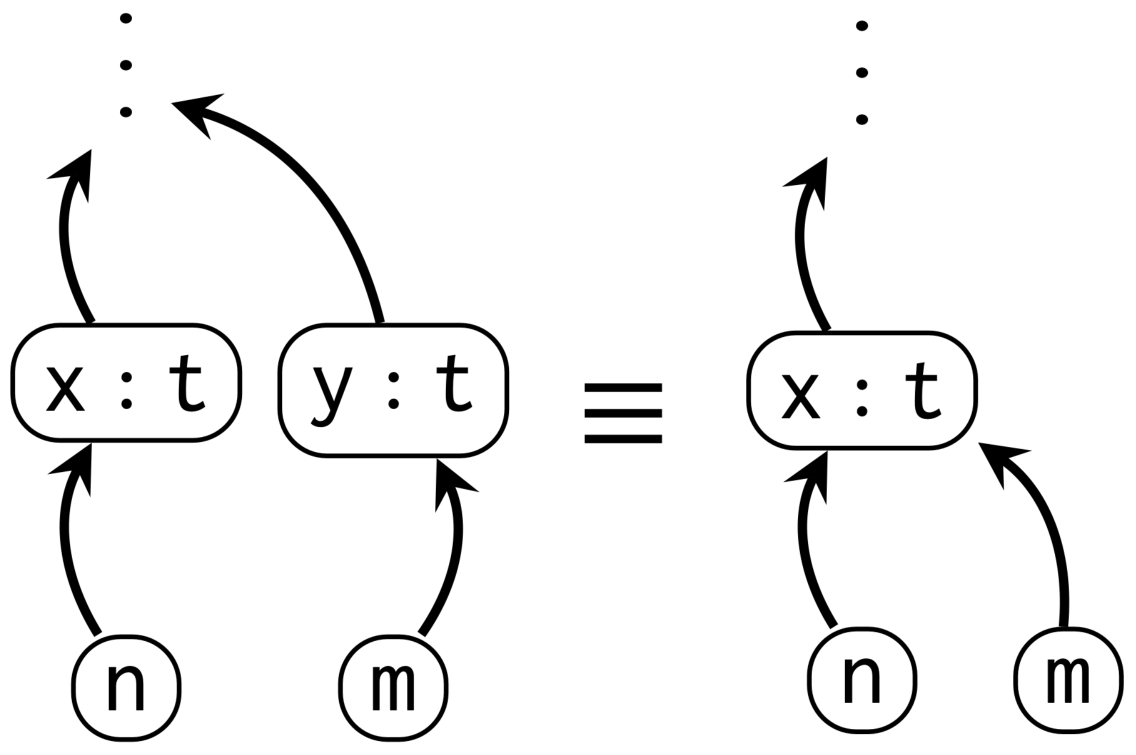
Destructive "***kill***" effects model type state and uniqueness (but add no new edges).



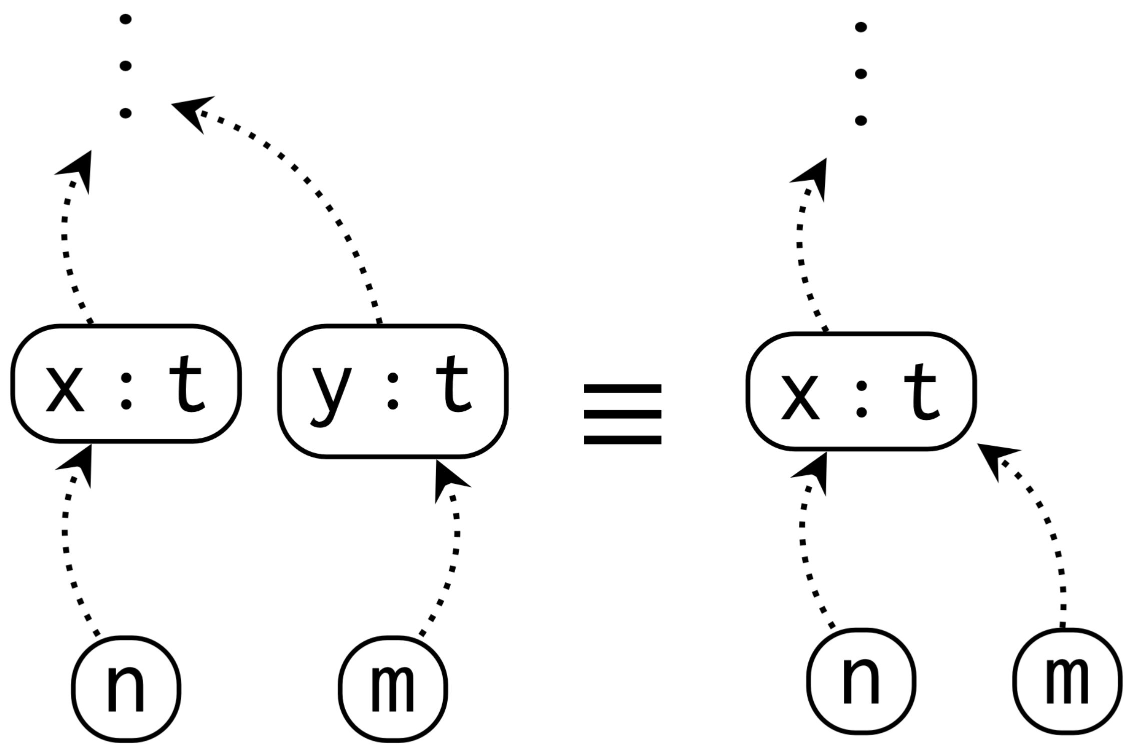
GRAPH-LEVEL OPTIMIZATIONS



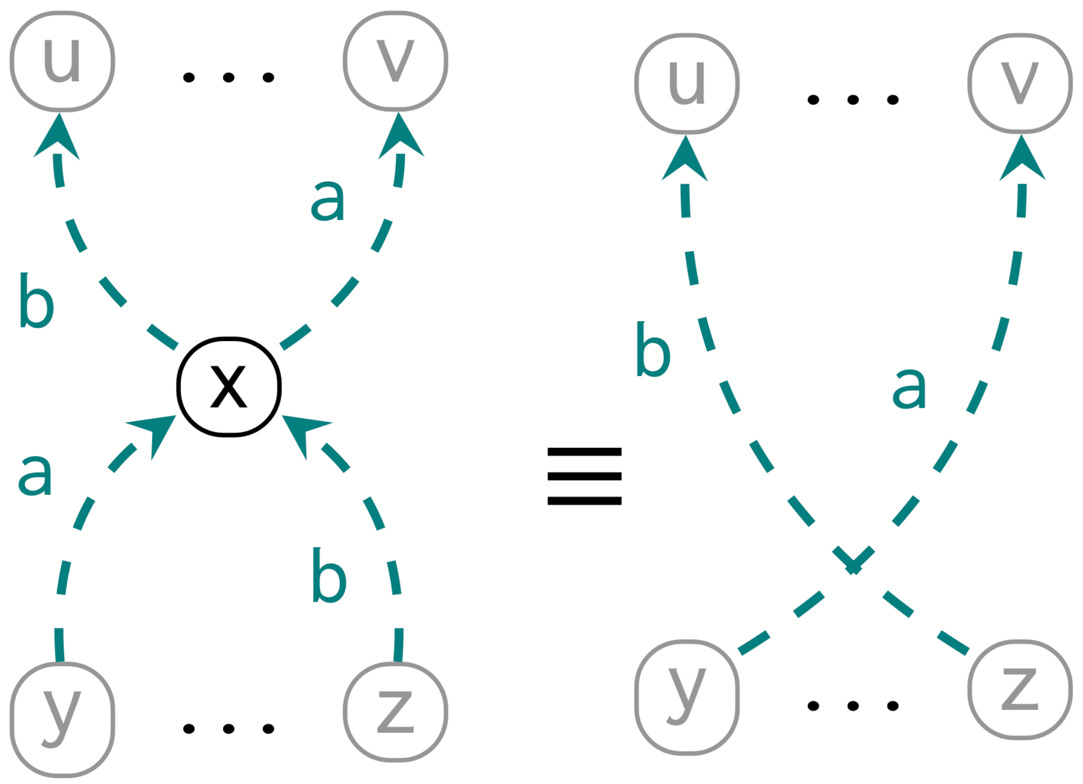
COMM



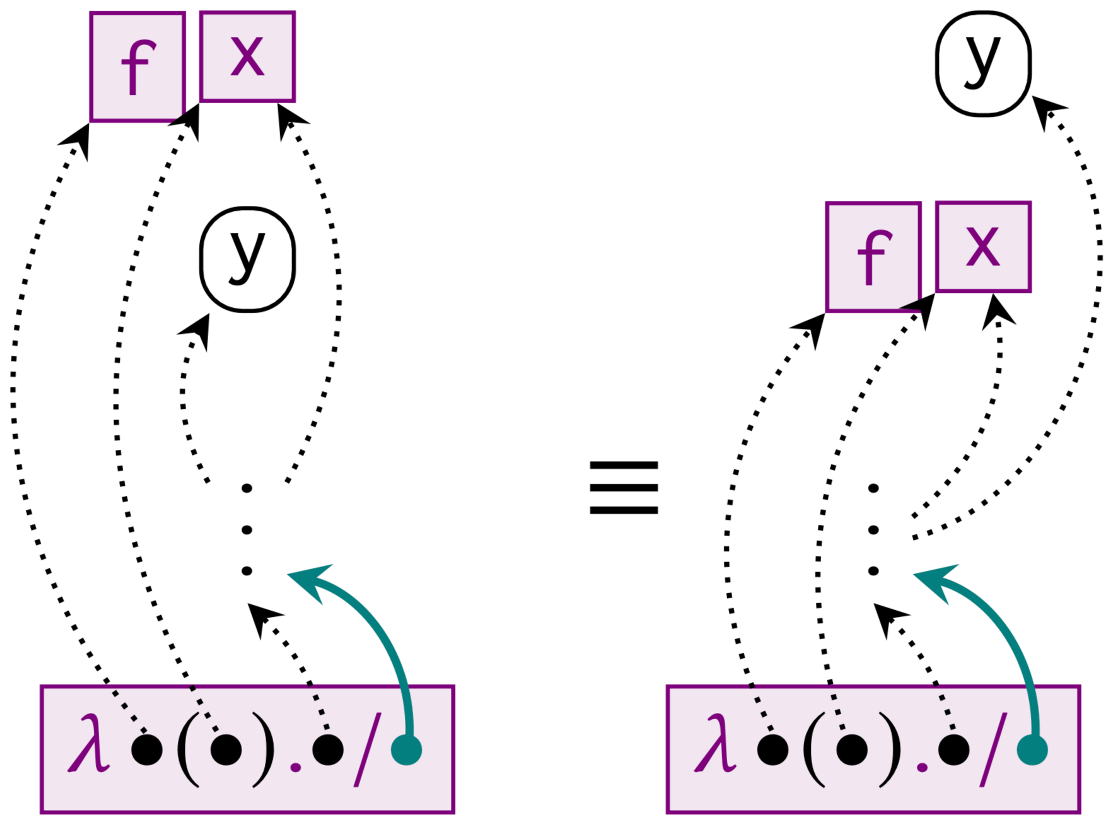
E-CSE



CSE



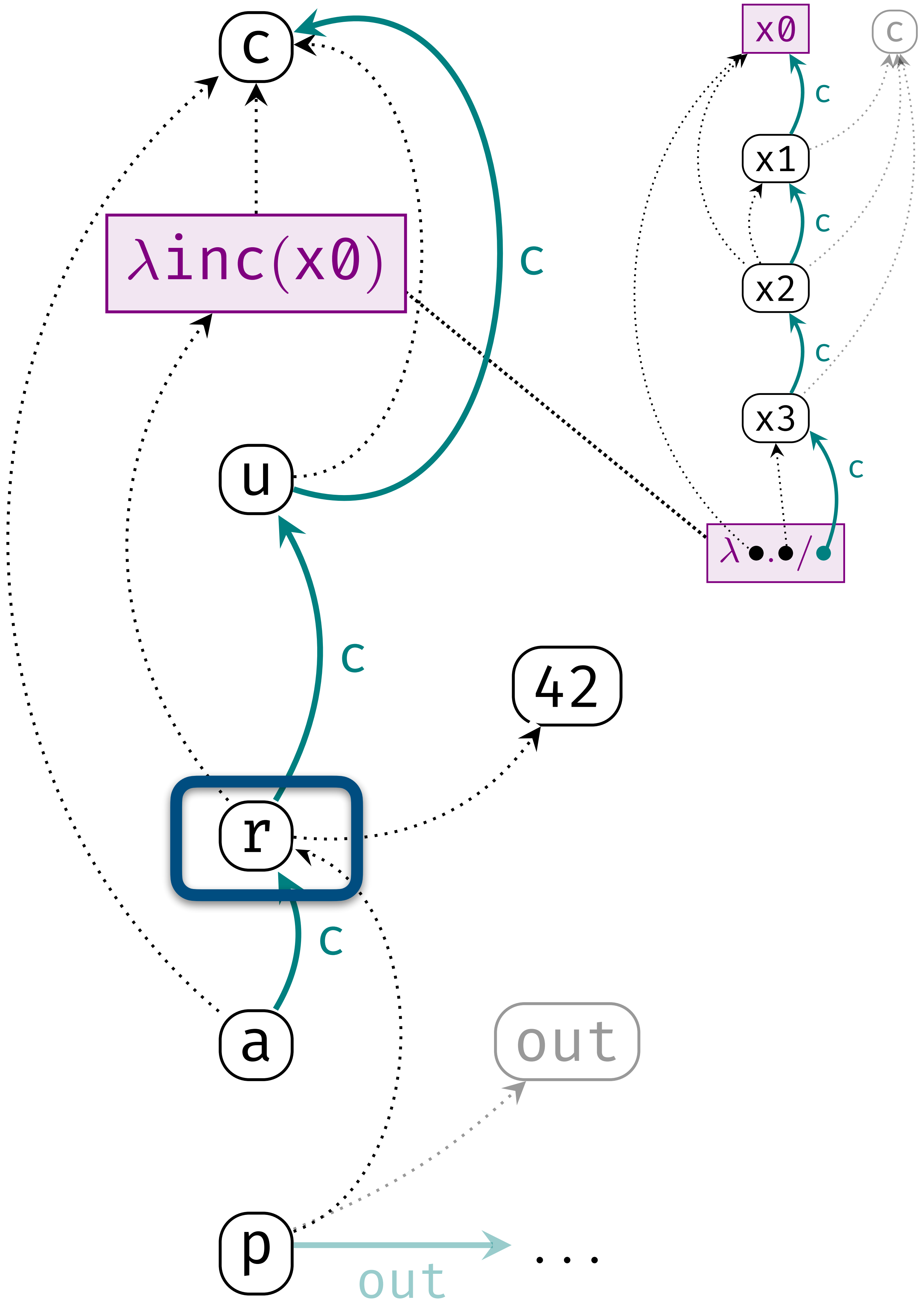
DCE



LAMBDA-HOIST

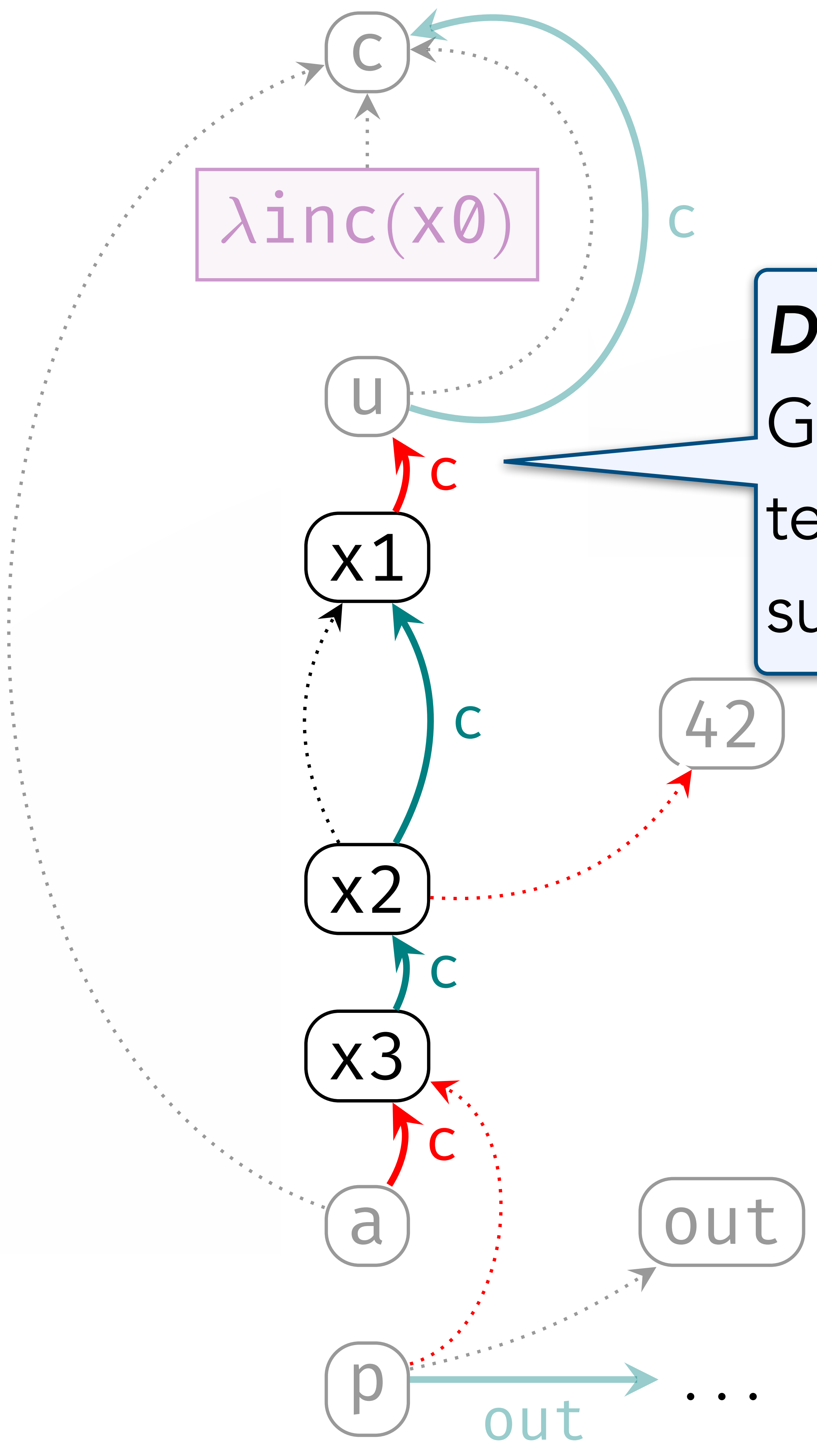
FUNCTION CALLS & INLINING

```
val c = new Ref(0)

def inc(x0: Int) = {
  val x1 = !c
  val x2 = c := x1 + x0
  val x3 = !c
  x3
}

val u = c := 21

val r = inc(42)

val a = c := 0
val p = println(out, r)
```



FUNCTION CALLS & INLINING

```
val c = new Ref(0)

def inc(x0: Int) = {
  val x1 = !c
  val x2 = c := x1 + x0
  val x3 = !c
  x3
}

val u = c := 21
val x1 = !c
val x2 = c := x1 + 42
val x3 = !c
val a = c := 0
val p = println(out, r)
```

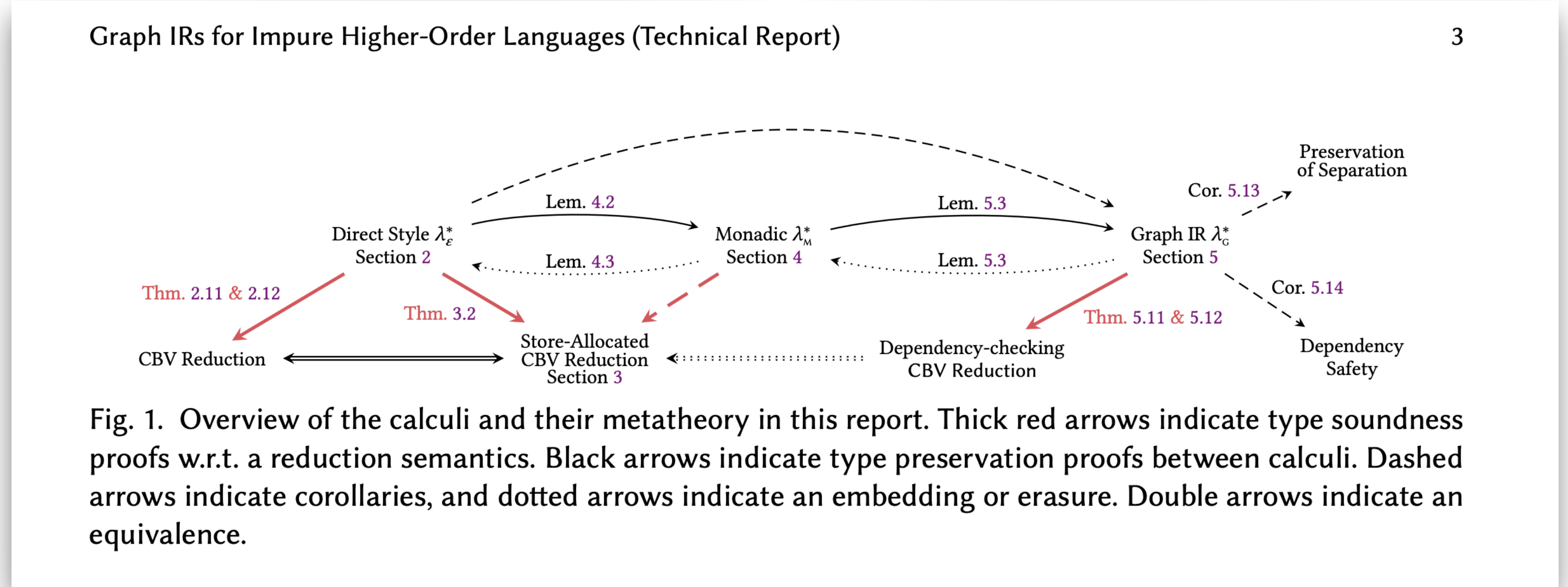


Dependency Rewiring
Graph-level analogue of term- and type-level substitution.

FORMAL METATHEORY

Proved Properties of the Core Graph IR:

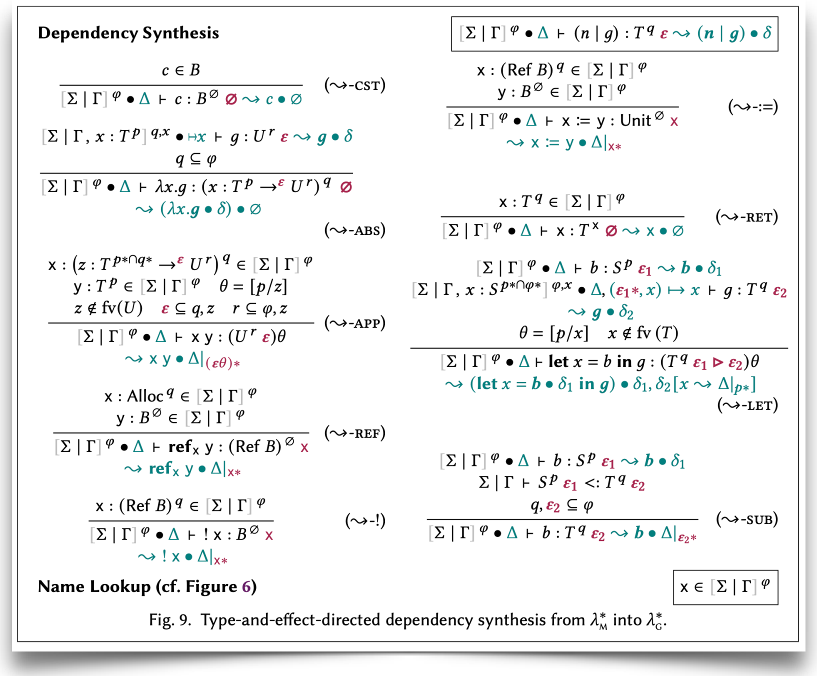
- ***End-to-end type-preserving translation*** from direct style lambda calculus with reachability types to graph terms.
- ***Type-and-Dependency safety***
  - Dependencies of well-typed graph terms respect the program's control flow.
- ***Soundness of equational graph-term transformations***, including β-equality.
  - Justifies that our ANF terms are really graphs.
  - Proof by contextual equivalence using logical relations.



Graph IRs for Impure Higher-Order Languages (Technical Report) 3

Direct Style λ^*_ε
Section 2
Lem. 4.2
Lem. 4.3
Monadic λ^*_{M}
Section 4
Lem. 5.3
Lem. 5.3
Graph IR λ^*_{G}
Section 5
Cor. 5.13
Preservation of Separation
Cor. 5.14
Dependency Safety
Thm. 2.11 & 2.12
Thm. 3.2
Thm. 5.11 & 5.12
CBV Reduction
Store-Allocated CBV Reduction Section 3
Dependency-checking CBV Reduction

Fig. 1. Overview of the calculi and their metatheory in this report. Thick red arrows indicate type soundness proofs w.r.t. a reduction semantics. Black arrows indicate type preservation proofs between calculi. Dashed arrows indicate corollaries, and dotted arrows indicate an embedding or erasure. Double arrows indicate an equivalence.



Dependency Synthesis $\boxed{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash (n \mid g) : T^{q}\ \varepsilon \rightsquigarrow (n \mid g) \bullet \delta}$

$$\frac{c \in B}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash c : B^{\varnothing}\ \varnothing \rightsquigarrow c \bullet \varnothing}\quad (\rightsquigarrow\text{-CST})$$

$$\frac{[\Sigma \mid \Gamma, x : T^{p}]^{q,x} \bullet \mapsto x \vdash g : U^{r}\ \varepsilon \rightsquigarrow g \bullet \delta \qquad q \subseteq \varphi}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash \lambda x.g : (x : T^{p} \to^{\varepsilon} U^{r})^{q}\ \varnothing \rightsquigarrow (\lambda x.g \bullet \delta) \bullet \varnothing}\quad (\rightsquigarrow\text{-ABS})$$

$$\frac{x : (z : T^{p*\cap q*} \to^{\varepsilon} U^{r})^{q} \in [\Sigma \mid \Gamma]^{\varphi} \quad y : T^{p} \in [\Sigma \mid \Gamma]^{\varphi} \quad \theta = [p/z] \quad z \notin \mathrm{fv}(U) \quad \varepsilon \subseteq q, z \quad r \subseteq \varphi, z}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash x\ y : (U^{r}\ \varepsilon)\theta \rightsquigarrow x\ y \bullet \Delta|_{(\varepsilon\theta)*}}\quad (\rightsquigarrow\text{-APP})$$

$$\frac{x : \mathsf{Alloc}^{q} \in [\Sigma \mid \Gamma]^{\varphi} \quad y : B^{\varnothing} \in [\Sigma \mid \Gamma]^{\varphi}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash \mathbf{ref}_{x}\ y : (\mathsf{Ref}\ B)^{\varnothing}\ x \rightsquigarrow \mathbf{ref}_{x}\ y \bullet \Delta|_{x*}}\quad (\rightsquigarrow\text{-REF})$$

$$\frac{x : (\mathsf{Ref}\ B)^{q} \in [\Sigma \mid \Gamma]^{\varphi}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash\ !x : B^{\varnothing}\ x \rightsquigarrow\ !x \bullet \Delta|_{x*}}\quad (\rightsquigarrow\text{-!})$$

$$\frac{x : (\mathsf{Ref}\ B)^{q} \in [\Sigma \mid \Gamma]^{\varphi} \quad y : B^{\varnothing} \in [\Sigma \mid \Gamma]^{\varphi}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash x := y : \mathsf{Unit}^{\varnothing}\ x \rightsquigarrow x := y \bullet \Delta|_{x*}}\quad (\rightsquigarrow\text{-:=})$$

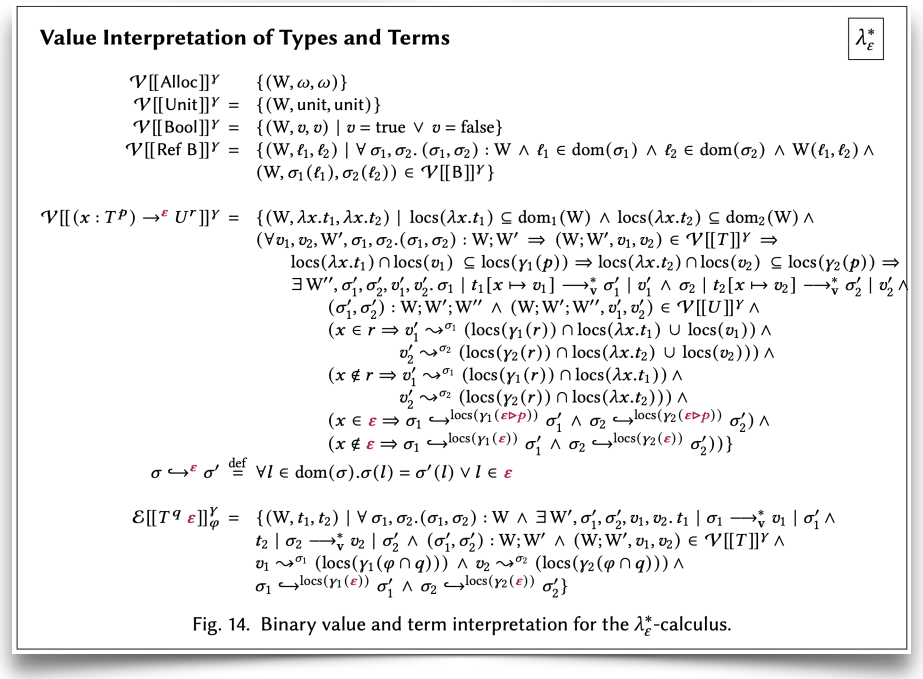
$$\frac{x : T^{q} \in [\Sigma \mid \Gamma]^{\varphi}}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash x : T^{x}\ \varnothing \rightsquigarrow x \bullet \varnothing}\quad (\rightsquigarrow\text{-RET})$$

$$\frac{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash b : S^{p}\ \varepsilon_1 \rightsquigarrow b \bullet \delta_1 \quad [\Sigma \mid \Gamma, x : S^{p*\cap\varphi*}]^{\varphi,x} \bullet \Delta, (\varepsilon_1*, x) \mapsto x \vdash g : T^{q}\ \varepsilon_2 \rightsquigarrow g \bullet \delta_2 \quad \theta = [p/x] \quad x \notin \mathrm{fv}(T)}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash \mathbf{let}\ x = b\ \mathbf{in}\ g : (T^{q}\ \varepsilon_1 \rhd \varepsilon_2)\theta \rightsquigarrow (\mathbf{let}\ x = b \bullet \delta_1\ \mathbf{in}\ g) \bullet \delta_1, \delta_2[x \rightsquigarrow \Delta|_{p*}]}\quad (\rightsquigarrow\text{-LET})$$

$$\frac{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash b : S^{p}\ \varepsilon_1 \rightsquigarrow b \bullet \delta_1 \quad \Sigma \mid \Gamma \vdash S^{p}\ \varepsilon_1 <: T^{q}\ \varepsilon_2 \quad q, \varepsilon_2 \subseteq \varphi}{[\Sigma \mid \Gamma]^{\varphi} \bullet \Delta \vdash b : T^{q}\ \varepsilon_2 \rightsquigarrow b \bullet \Delta|_{\varepsilon_2*}}\quad (\rightsquigarrow\text{-SUB})$$

Name Lookup (cf. Figure 6) $\boxed{x \in [\Sigma \mid \Gamma]^{\varphi}}$

Fig. 9. Type-and-effect-directed dependency synthesis from λ^*_{M} into λ^*_{G}.



Value Interpretation of Types and Terms $\boxed{\lambda^*_\varepsilon}$

$$\begin{aligned}
\mathcal{V}[[\mathsf{Alloc}]]^{\gamma}\ & \{(W, \omega, \omega)\} \\
\mathcal{V}[[\mathsf{Unit}]]^{\gamma} &= \{(W, \mathsf{unit}, \mathsf{unit})\} \\
\mathcal{V}[[\mathsf{Bool}]]^{\gamma} &= \{(W, v, v) \mid v = \mathsf{true} \lor v = \mathsf{false}\} \\
\mathcal{V}[[\mathsf{Ref\ B}]]^{\gamma} &= \{(W, \ell_1, \ell_2) \mid \forall \sigma_1, \sigma_2.\ (\sigma_1, \sigma_2) : W \land \ell_1 \in \mathrm{dom}(\sigma_1) \land \ell_2 \in \mathrm{dom}(\sigma_2) \land W(\ell_1, \ell_2) \land \\
&\qquad (W, \sigma_1(\ell_1), \sigma_2(\ell_2)) \in \mathcal{V}[[\mathsf{B}]]^{\gamma}\} \\
\mathcal{V}[[(x : T^{p}) \to^{\varepsilon} U^{r}]]^{\gamma} &= \{(W, \lambda x.t_1, \lambda x.t_2) \mid \mathrm{locs}(\lambda x.t_1) \subseteq \mathrm{dom}_1(W) \land \mathrm{locs}(\lambda x.t_2) \subseteq \mathrm{dom}_2(W) \land \\
&\qquad (\forall v_1, v_2, W', \sigma_1, \sigma_2.(\sigma_1, \sigma_2) : W; W' \Rightarrow (W; W', v_1, v_2) \in \mathcal{V}[[T]]^{\gamma} \Rightarrow \\
&\qquad\quad \mathrm{locs}(\lambda x.t_1) \cap \mathrm{locs}(v_1) \subseteq \mathrm{locs}(\gamma_1(p)) \Rightarrow \mathrm{locs}(\lambda x.t_2) \cap \mathrm{locs}(v_2) \subseteq \mathrm{locs}(\gamma_2(p)) \Rightarrow \\
&\qquad\quad \exists W'', \sigma_1', \sigma_2', v_1', v_2'.\ \sigma_1 \mid t_1[x \mapsto v_1] \longrightarrow^*_{\mathrm{v}} \sigma_1' \mid v_1' \land \sigma_2 \mid t_2[x \mapsto v_2] \longrightarrow^*_{\mathrm{v}} \sigma_2' \mid v_2' \land \\
&\qquad\qquad (\sigma_1', \sigma_2') : W; W'; W'' \land (W; W'; W'', v_1', v_2') \in \mathcal{V}[[U]]^{\gamma} \land \\
&\qquad\qquad (x \in r \Rightarrow v_1' \rightsquigarrow^{\sigma_1} (\mathrm{locs}(\gamma_1(r)) \cap \mathrm{locs}(\lambda x.t_1) \cup \mathrm{locs}(v_1)) \land \\
&\qquad\qquad\qquad v_2' \rightsquigarrow^{\sigma_2} (\mathrm{locs}(\gamma_2(r)) \cap \mathrm{locs}(\lambda x.t_2) \cup \mathrm{locs}(v_2))) \land \\
&\qquad\qquad (x \notin r \Rightarrow v_1' \rightsquigarrow^{\sigma_1} (\mathrm{locs}(\gamma_1(r)) \cap \mathrm{locs}(\lambda x.t_1)) \land \\
&\qquad\qquad\qquad v_2' \rightsquigarrow^{\sigma_2} (\mathrm{locs}(\gamma_2(r)) \cap \mathrm{locs}(\lambda x.t_2))) \land \\
&\qquad\qquad (x \in \varepsilon \Rightarrow \sigma_1 \hookrightarrow^{\mathrm{locs}(\gamma_1(\varepsilon \rhd p))} \sigma_1' \land \sigma_2 \hookrightarrow^{\mathrm{locs}(\gamma_2(\varepsilon \rhd p))} \sigma_2') \land \\
&\qquad\qquad (x \notin \varepsilon \Rightarrow \sigma_1 \hookrightarrow^{\mathrm{locs}(\gamma_1(\varepsilon))} \sigma_1' \land \sigma_2 \hookrightarrow^{\mathrm{locs}(\gamma_2(\varepsilon))} \sigma_2'))\} \\
\sigma \hookrightarrow^{\varepsilon} \sigma' &\overset{\mathrm{def}}{=} \forall l \in \mathrm{dom}(\sigma).\sigma(l) = \sigma'(l) \lor l \in \varepsilon \\
\mathcal{E}[[T^{q}\ \varepsilon]]^{\gamma}_{\varphi} &= \{(W, t_1, t_2) \mid \forall \sigma_1, \sigma_2.(\sigma_1, \sigma_2) : W \land \exists W', \sigma_1', \sigma_2', v_1, v_2.\ t_1 \mid \sigma_1 \longrightarrow^*_{\mathrm{v}} v_1 \mid \sigma_1' \land \\
&\qquad t_2 \mid \sigma_2 \longrightarrow^*_{\mathrm{v}} v_2 \mid \sigma_2' \land (\sigma_1', \sigma_2') : W; W' \land (W; W', v_1, v_2) \in \mathcal{V}[[T]]^{\gamma} \land \\
&\qquad v_1 \rightsquigarrow^{\sigma_1} (\mathrm{locs}(\gamma_1(\varphi \cap q))) \land v_2 \rightsquigarrow^{\sigma_2} (\mathrm{locs}(\gamma_2(\varphi \cap q))) \land \\
&\qquad \sigma_1 \hookrightarrow^{\mathrm{locs}(\gamma_1(\varepsilon))} \sigma_1' \land \sigma_2 \hookrightarrow^{\mathrm{locs}(\gamma_2(\varepsilon))} \sigma_2'\}
\end{aligned}$$

Fig. 14. Binary value and term interpretation for the λ^*_ε-calculus.

Our results have ***strong formal backing***, based on the fully mechanized reachability types metatheory in Coq:

https://github.com/tiarkRompf/reachability

For full details:
Check out our ***60-page companion paper!***

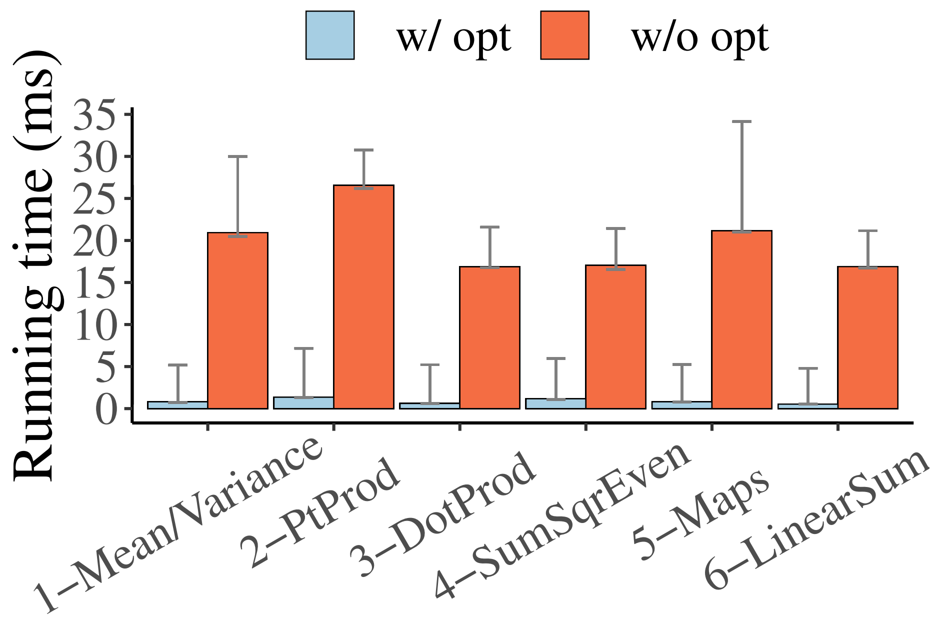
http://arxiv.org/abs/2309.08118

EVALUATION

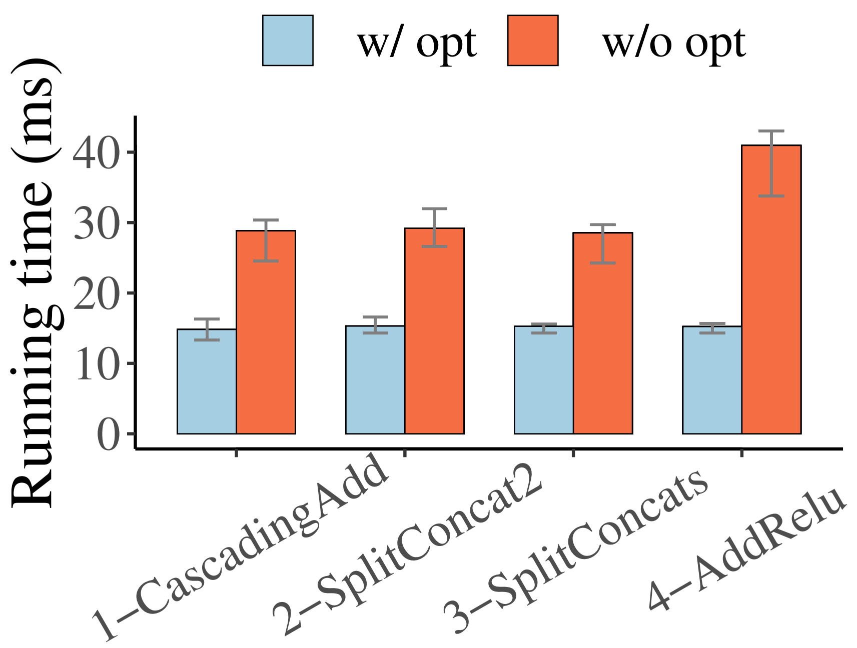
Case Studies in Scala LMS + Graph IR

Tensor Fusion CPU/GPU

[Wang et al. Big Data'19, ICFP'19]



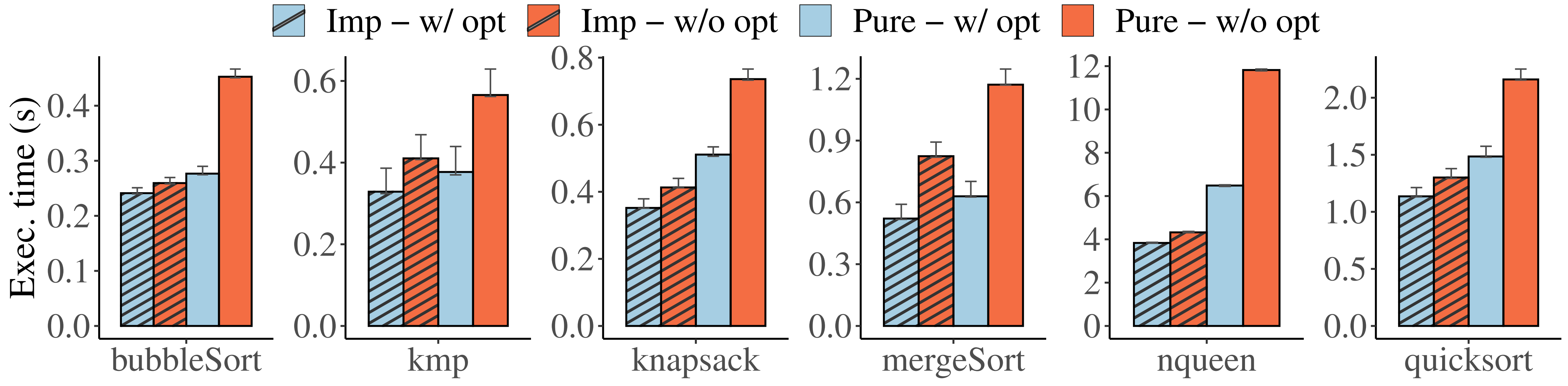
Tensor Loop Fusion - max. 21x



CUDA Kernel Fusion - max. 2x

Symbolic-Execution Compiler

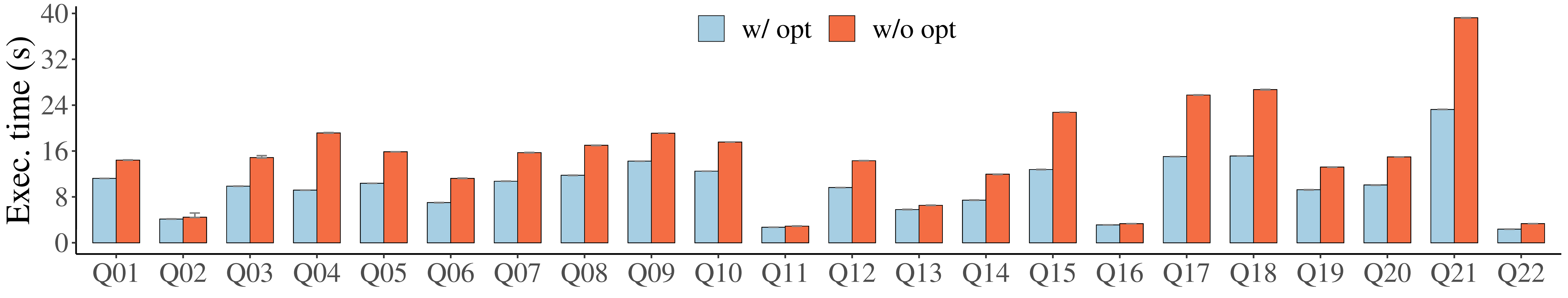
[Wei et al. OOPSLA'20, FSE'21, ICSE'23]



Imperative reimplementation - max. 3.1x

SQL-Query Compiler

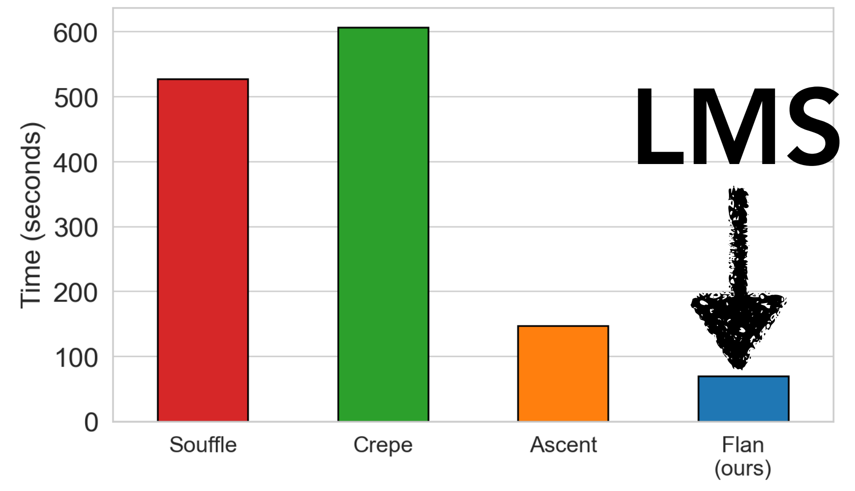
[Essertel et al. OSDI'18; Rompf and Amin ICFP'15; Tahboub et al. SIGMOD'18]



TPC-H Benchmark - max. 1.8x

Datalog Compiler NEW

[Abeysinghe et al., conditionally accepted at POPL'24]



E.g., Points-to Analysis (7.1x faster than Souffle) & more real-world stuff

SUMMARY & CONTRIBUTIONS

Finally, a Graph IR for Lambda and Effects!

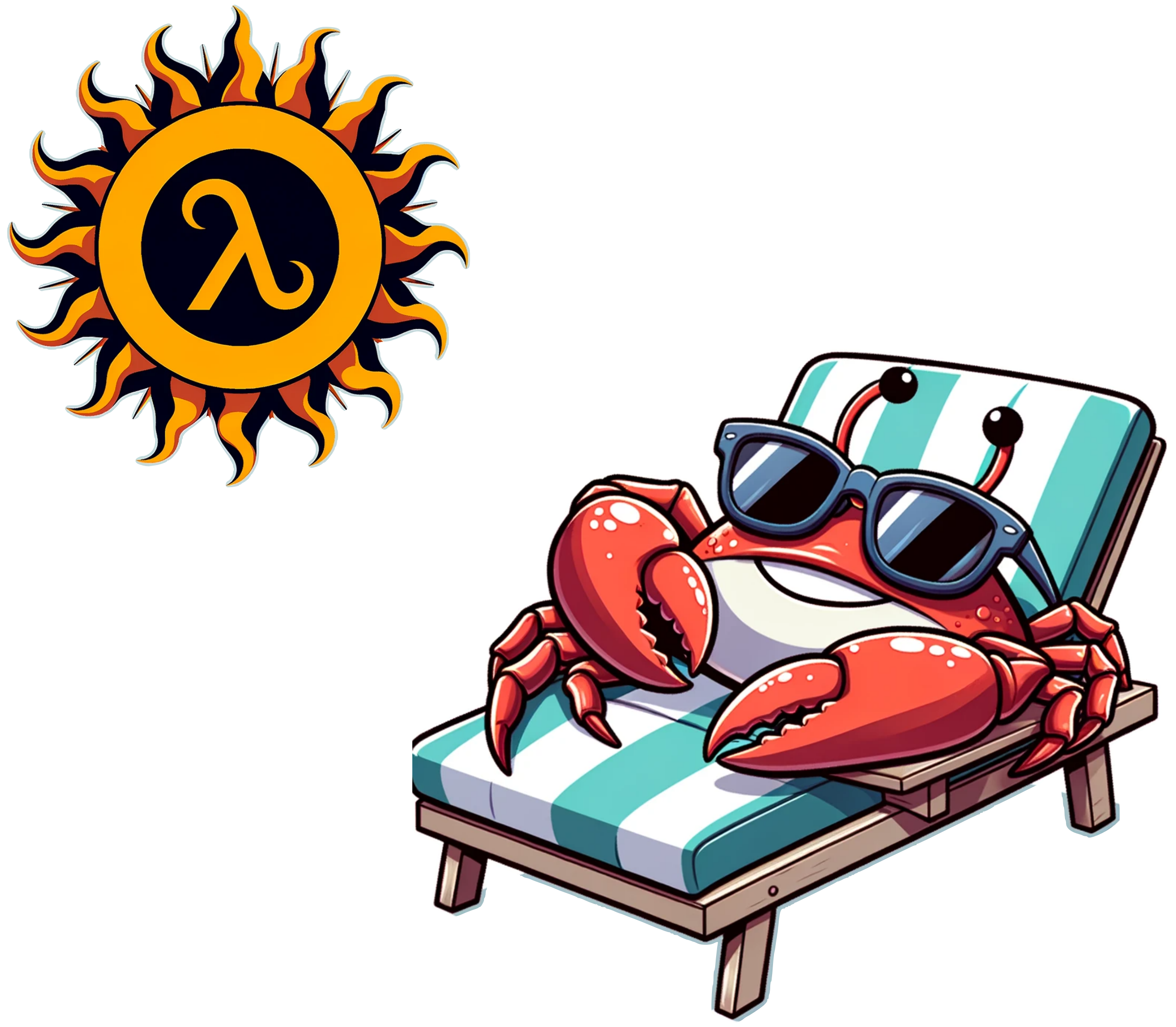
- We unlock ***aggressive, affordable, and global optimizations*** for impure functional languages.
- ***Ownership types, dependent types, and type inference*** yield good dependencies!
- ***Seamless code motion from lambdas in the graph!***
- Correctness backed by ***strong formal metatheory***. Full details at `http://arxiv.org/abs/2309.08118`

More Goodness in the Paper:

- ***From graphs back to trees:*** Code generation and code motion. Basic algorithm + dead code elimination + frequency estimation. Full details at `http://arxiv.org/abs/2309.08118`
- ***Higher-order program optimization:*** Restricted cases out of the box for lambda lifting and super-beta inlining. Need flow analyses for the full deal.

Artifacts:

- Scala LMS with Effect Dependencies
  `https://github.com/tiarkRompf/lms-clean`
- Mechanized metatheory of the reachability types universe + mini LMS/Graph IR prototype
  `https://github.com/tiarkRompf/reachability`



Thanks, and enjoy Cascais!