Modular, Higher-Order Cardinality Analysis in Theory and Practice

Ilya Sergey   Dimitrios Vytiniotis   Simon Peyton Jones

institute imdea software   Microsoft Research

POPL 2014

A story of
three program optimisations

Optimisation 1

```
f1, f2 ::  [Int] -> Int
```

Which function is better to run?

Better

```
f1 xs = let ys = map costly xs
        in squash (\n. sum (map (+ n) ys))
```

if invoked *more than once* by `squash`

```
f2 xs = squash (\n. sum (map (+ n) (map costly xs)))
```

Better

if invoked *at most* once by `squash`

How many times

a function is called?

(call cardinality)

Optimisation 2

“worker-wrapper” split

```
f x = case x of (p,q) -> <cbody>
```

Optimisation 2

"worker-wrapper" split

"wrapper", usually inlined on-site

```
f x = case x of (p,q) -> fw p q

fw p q = <cbody>
```

"worker"

Optimisation 2

“worker-wrapper” split

What if `q` is *never* used in `<cbody>`?

```
f x = case x of (p,q) -> fw p

fw p   = <cbody>
```

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

g y = f y (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)

call demand

Usage demands	$d \quad ::=$	$\boxed{C^n(d)} \mid U(d_1^\dagger, d_2^\dagger) \mid U$
Cardinality demands	$d^\dagger \quad ::=$	$A \mid n * d$
Usage cardinalities	$n \quad ::=$	$1 \mid \omega$

tuple demand

Usage demands	d	$::=$	$C^n(d) \mid U(d_1^\dagger, d_2^\dagger) \mid U$
Cardinality demands	$d^\dagger$	$::=$	$A \mid n * d$
Usage cardinalities	n	$::=$	$1 \mid \omega$

general demand

Usage demands $d \quad ::= \quad C^n(d) \mid U(d_1^\dagger, d_2^\dagger) \mid \boxed{U}$

Cardinality demands $d^\dagger \quad ::= \quad A \mid n * d$

Usage cardinalities $n \quad ::= \quad 1 \mid \omega$

Usage demands $d \quad ::= \quad C^n(d) \mid U(d_1^\dagger, d_2^\dagger) \mid U$

absent value

Cardinality demands $d^\dagger \quad ::= \quad \boxed{A} \mid n * d$

Usage cardinalities $n \quad ::= \quad 1 \mid \omega$

Usage demands	d	$::=$	$C^n(d) \mid U(d_1^\dagger, d_2^\dagger) \mid U$
			used at most n times
Cardinality demands	$d^\dagger$	$::=$	$A \mid n * d$
Usage cardinalities	n	$::=$	$1 \mid \omega$

Usage Types

(*how* a function uses its arguments)

$\texttt{wurble1 ::}\ \boxed{\omega * U} \to C^{\omega}(C^{1}(U)) \to \bullet$

```
wurble1 a g = g 2 [a] + g 3 [a]
```

```
wurble1 ::
```
$\omega * U \to C^{\omega}(C^{1}(U)) \to \bullet$

```
wurble1 a g = g 2 a + g 3 a
```

`wurble2 ::` $\omega * U \to C^1(C^\omega(U)) \to \bullet$

```
wurble2 a g = sum (map (g a) [1..1000])
```

wurble2 :: $\omega * U \to C^1(C^\omega(U)) \to \bullet$

```
wurble2 a g = sum (map (g a) [1..1000])
```

```
f ::
```
$$1{*}U(1{*}U, A) \to \bullet$$

```
f x = case x of (p, q) -> p + 1
```

Usage type
depends on a usage context!

(*result* demand determines *argument* demands)

Backwards Analysis

Infers demand type basing on a context

$$P \mapsto e \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle$$

$$P \mapstochar\!\!\blacktriangleright e \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle$$

- 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

- τ - *demand type*, usages that e places on its arguments

- φ - *fv-usage*, usages that e places on its free variables

$$C^1(U)$$

$$e = \lambda x\ .\ \texttt{case}\ x\ \texttt{of}\ (p,\, q) \to (p,\, f\ \texttt{True})$$

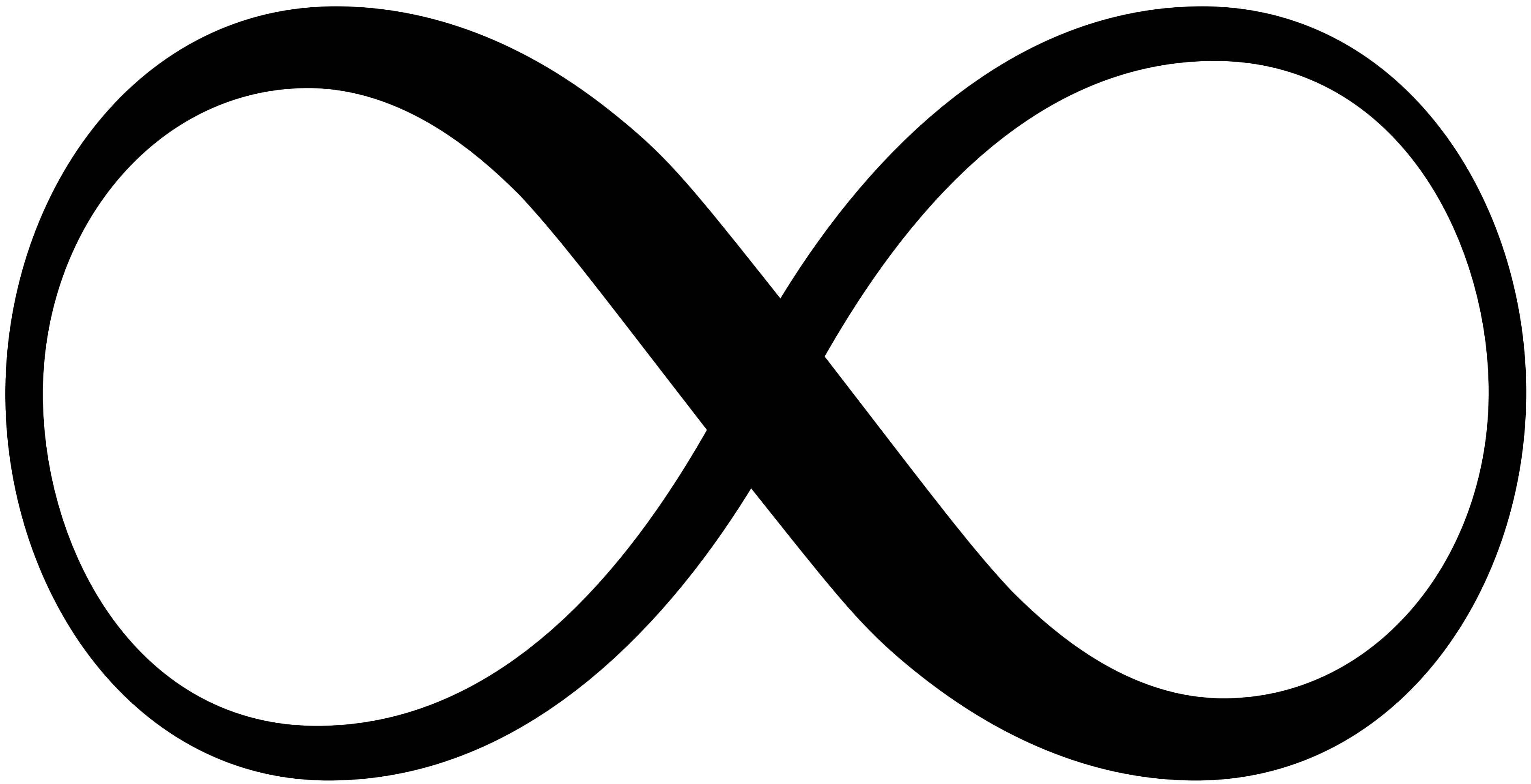
$$e = \lambda x \;.\; \texttt{case}\ x\ \texttt{of}\ (p, q) \rightarrow (p, f\ \texttt{True})$$

$$\epsilon \mapsto e \downarrow C^1(U) \Rightarrow \langle \underbrace{1 * U(\omega * U, A) \rightarrow \bullet}_{\tau}\, ; \underbrace{\{f \mapsto 1 * C^1(U)\}}_{\varphi} \rangle$$

Each function is a

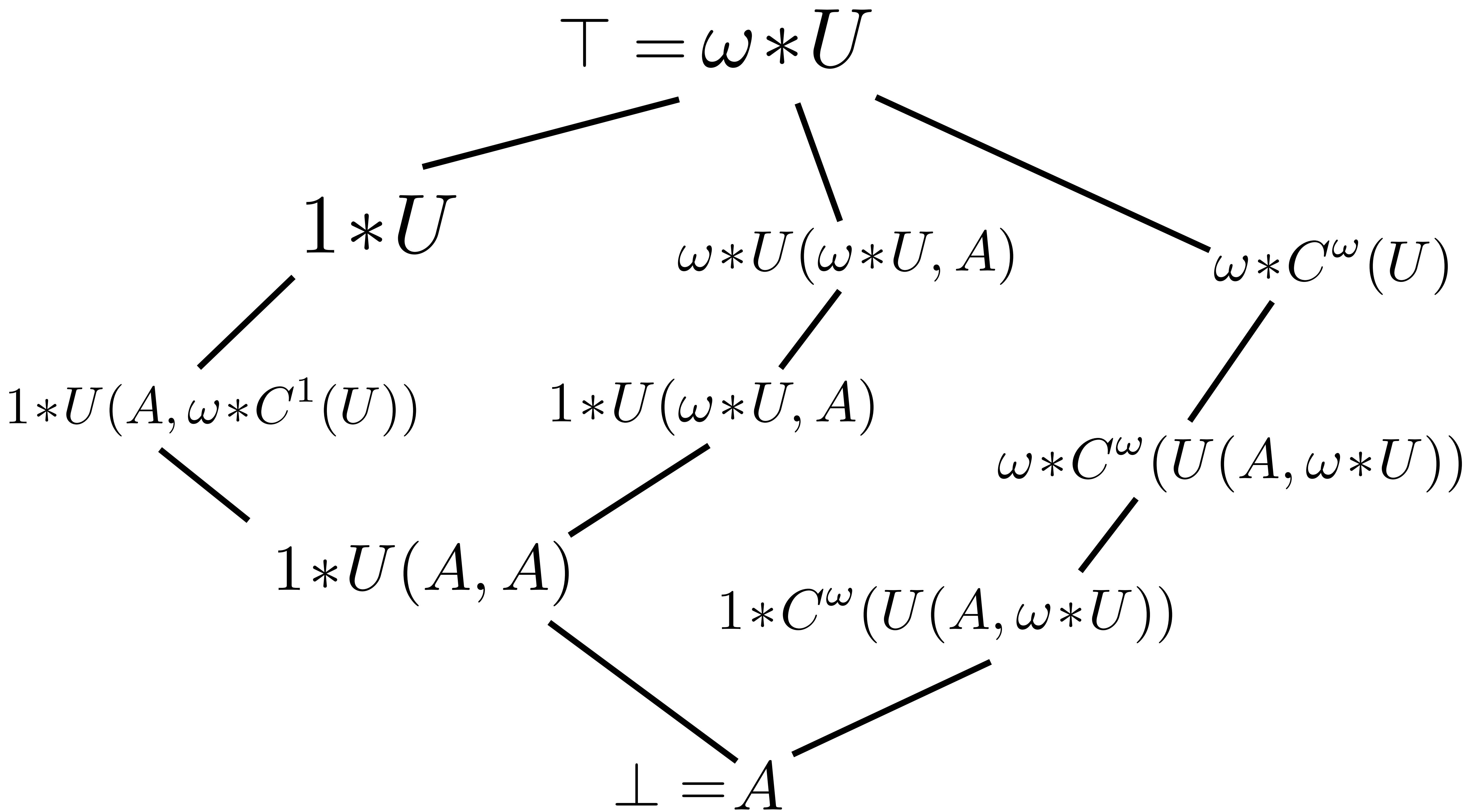
backwards demand transformer

it transforms a *context* demand to *argument* demands and *fv*-demands.



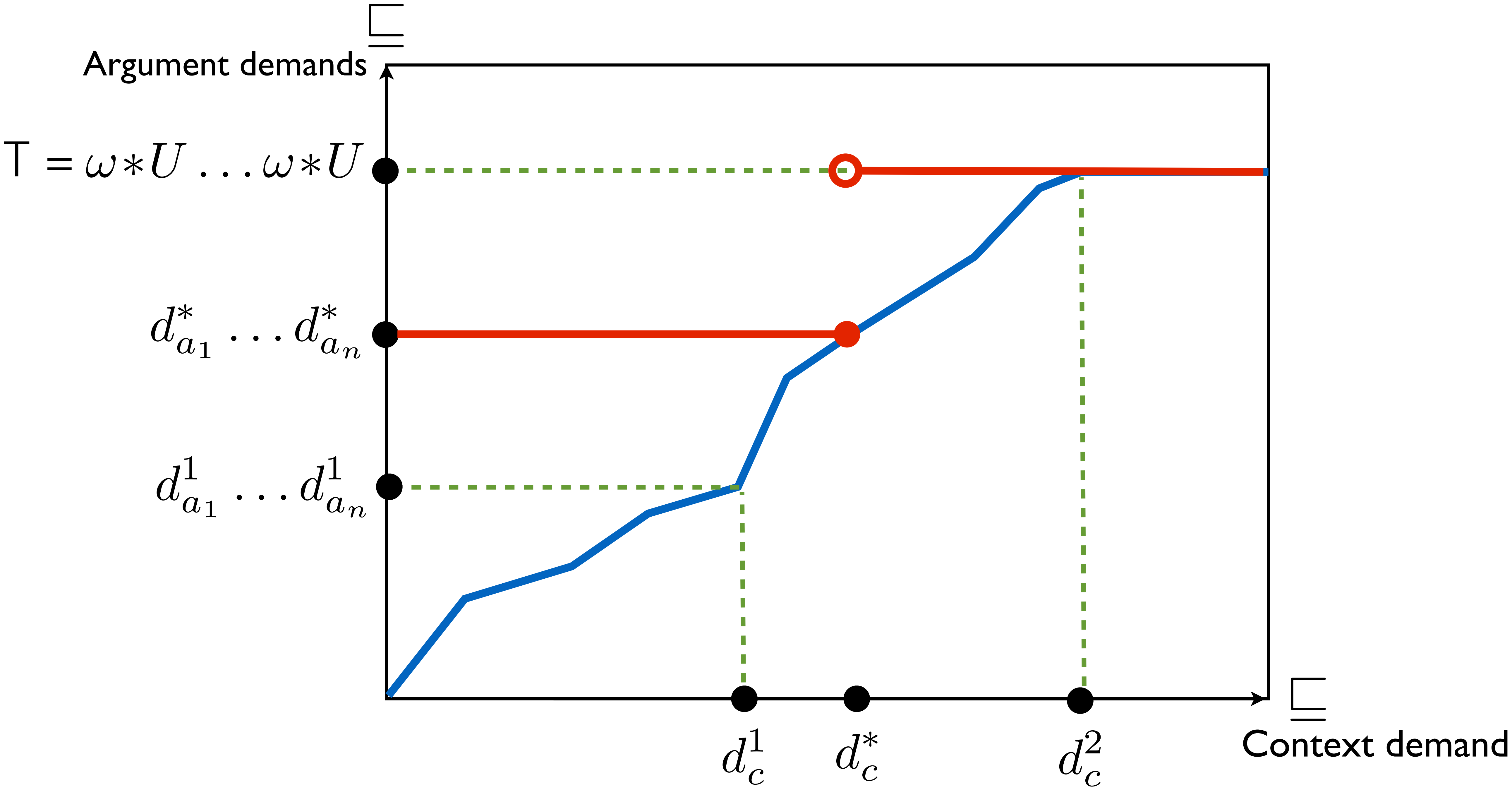
We cannot compute best argument demands for *all* contexts: need to *approximate*.

Demand Lattice



Each function is
a monotone backwards demand transformer.

Exploiting demand monotonicity



Analysis-based annotations

$$P \vdash\!\!\blacktriangleright e \downarrow d \Rightarrow \langle \tau \,;\, \varphi \rangle$$

Elaboration

$$P \mapstochar\!\blacktriangleright e \downarrow d \Rightarrow \langle \tau \; ; \varphi \rangle \; \rightsquigarrow \mathsf{e}$$

- **let**-bindings in e are annotated with $\mathbf{m} \in \{\mathbf{0}, \mathbf{1}, \boldsymbol{\omega}\}$ to indicate how often the let binding is evaluated;
- Each Lambda $\boldsymbol{\lambda}^{n}\mathbf{x}\ .\mathbf{e}_{l}$ in e carries an annotation $\mathbf{n} \in \{\mathbf{1}, \boldsymbol{\omega}\}$ to indicate how often the lambda is called.

$$\epsilon \mapsto \texttt{let}\ f = \lambda x.\lambda y.\ x\ \texttt{True}\ \texttt{in}\ f\ p\ q \ \downarrow C^1(U)$$

$$\Rightarrow \langle \bullet; \{p \mapsto 1 * C^1(U), q \mapsto A\}\rangle$$

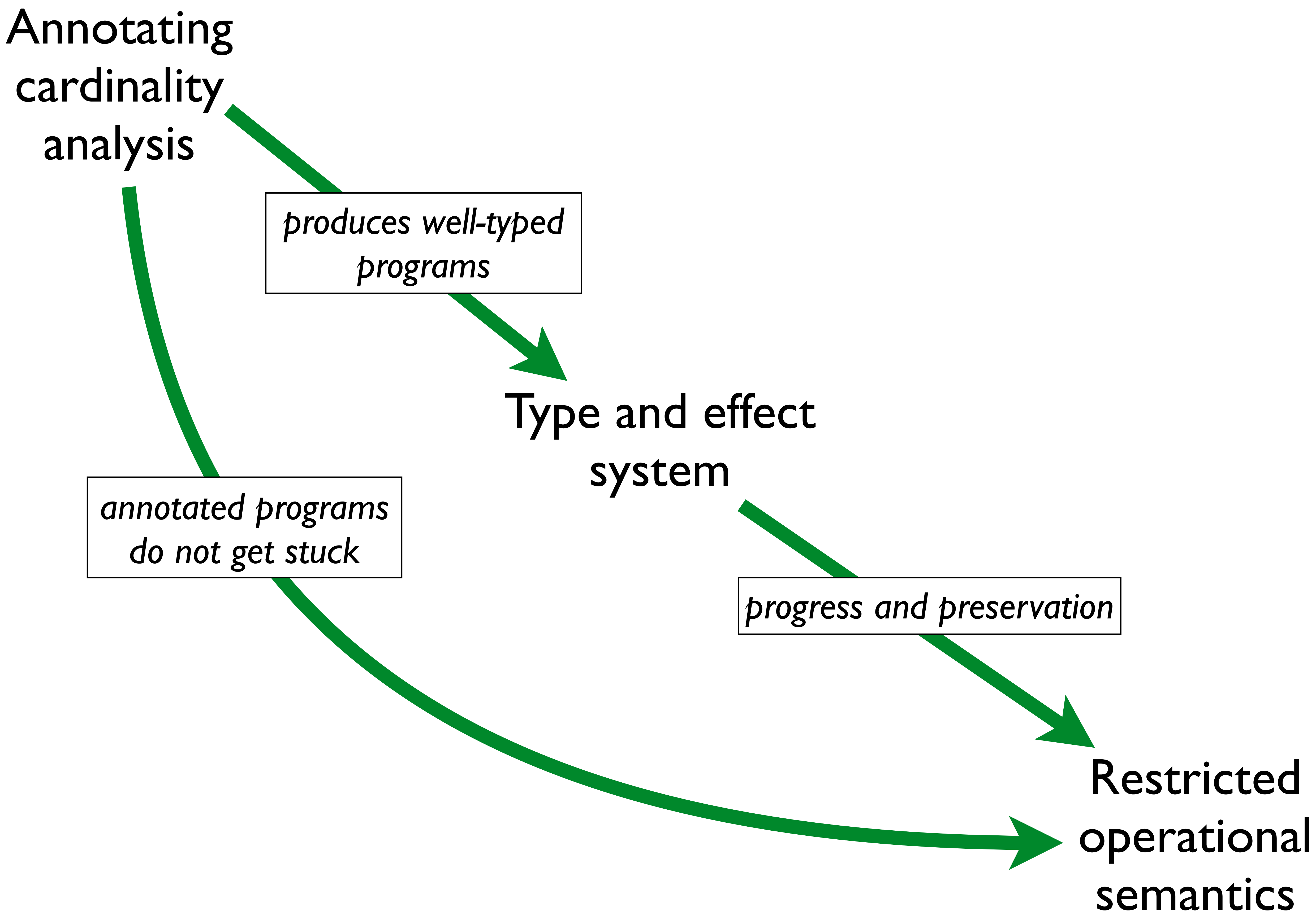
$$\rightsquigarrow$$

$$\texttt{let}\ f \overset{1}{=} \lambda^1 x.\lambda^1 y.\ x\ \texttt{True}\ \texttt{in}\ f\ p\ q$$

Soundness

Restricted operational semantics

(makes sure that the annotations are respected)


Annotating cardinality analysis
produces well-typed programs
Type and effect system
annotated programs do not get stuck
progress and preservation
Restricted operational semantics

Cardinality-enabled optimisations

1. Let-in floating optimisation

$$\texttt{let}\ z \overset{m_1}{=} \mathtt{e}_1\ \texttt{in}\ (\texttt{let}\ f \overset{m_2}{=} \lambda^1 x\ .\ \mathtt{e}\ \texttt{in}\ \mathtt{e}_2)$$

$$\texttt{let}\ z \stackrel{m_1}{=} \mathtt{e}_1\ \texttt{in}\ (\texttt{let}\ f \stackrel{m_2}{=} \lambda^{1} x\ .\ \mathtt{e}\ \texttt{in}\ \mathtt{e}_2)$$

$$\Longrightarrow \quad \texttt{let}\ f \stackrel{m_2}{=} \lambda^{1} x\ .\ (\texttt{let}\ z \stackrel{m_1}{=} \mathtt{e}_1\ \texttt{in}\ \mathtt{e})\ \texttt{in}\ \mathtt{e}_2,$$

for any m_1, m_2 and $z \notin FV(\mathtt{e}_2)$.

Improvement Theorem 1

Let-in floating
does not increase the number
of execution steps.

2. Smart execution

$$e_1$$

Optimised CBN Machine

Sestoft:JFP97

$$\langle \mathsf{H}_1, \mathsf{e}_1, \mathsf{S}_1 \rangle \Longrightarrow \ \ldots \ \Longrightarrow \langle \mathsf{H}_n, \mathsf{e}_n, \mathsf{S}_n \rangle$$

- ***1***-annotated bindings are *not* memoised;
- ***0***-annotated bindings are *skipped*.

Improvement Theorem 2

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`

*

Program	Synt. λ[1]	Synt. Thnk[1]	RT Thnk[1]
`anna`	4.0%	7.2%	2.9%
`bspt`	5.0%	15.4%	1.5%
`cacheprof`	7.6%	11.9%	5.1%
`calendar`	5.7%	0.0%	0.2%
`constraints`	2.0%	3.2%	4.5%
... and 72 more programs			
Arithmetic mean	10.3%	12.6%	5.5%

* as linked and run with libraries

Results on `nofib`

Program	Allocs		Runtime	
	No hack	Hack	No hack	Hack
`anna`	-2.1%	-0.2%	+0.1%	-0.0%
`bspt`	-2.2%	-0.0%	-0.0%	+0.0%
`cacheprof`	-7.9%	-0.6%	-6.1%	-5.0%
`calendar`	-9.2%	+0.2%	-0.0%	-0.0%
`constraints`	-0.9%	-0.0%	-1.2%	-0.2%
... and 72 more programs				
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%

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

Compiling with optimised GHC

- We compiled GHC *itself* with cardinality optimisations;
- Then we measured improvement in *compilation runtimes*.

Program	LOC	GHC Alloc Δ		GHC RT Δ	
		No hack	Hack	No hack	Hack
`anna`	5740	-1.6%	-1.5%	-0.8%	-0.4%
`cacheprof`	1600	-1.7%	-0.4%	-2.3%	-1.8%
`fluid`	1579	-1.9%	-1.9%	-2.8%	-1.6%
`gamteb`	1933	-0.5%	-0.1%	-0.5%	-0.1%
`parser`	2379	-0.7%	-0.2%	-2.6%	-0.6%
`veritas`	4674	-1.4%	-0.3%	-4.5%	-4.1%

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!