Cardinality Analysis and its Applications: from Glasgow Haskell Compiler to Sharded Blockchains

Ilya Sergey

YaleNUSCollege

NUS
National University of Singapore

Part I
Cardinality Analysis for Haskell Programs

Joint work with Dimitrios Vytiniotis and Simon Peyton Jones, presented at POPL'14

Optimising compiler in a nutshell

$$p_1$$

Optimising compiler in a nutshell

$$\begin{tikzcd}
p_1 \arrow[r] \arrow[d] & c_1 \\
p_2 \arrow[r] & c_2
\end{tikzcd}$$

$$c_2 \simeq c_1$$

$$c_2 \preceq c_1$$

$$
\begin{tikzcd}
p_1 \arrow[r] \arrow[d, "\mathcal{A}(p_1)"'] & c_1 \\
p_2 \arrow[r] & c_2
\end{tikzcd}
$$

Annotating static analysis

$$\mathcal{A}$$

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

Optimisation 1

```
squash1, squash2 :: (Int -> Int) -> Int

squash1 k = sum (map k [1..10])

squash2 k = 2 * (k 0)

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

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

Need to know:
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 memoisation

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

tuple demand

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

general demand

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

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

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

Usage Types

(*how* a function uses its arguments)

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

$$\texttt{wurble1 a g = g 2}\ \boxed{\texttt{a}}\ \texttt{+ g 3}\ \boxed{\texttt{a}}$$

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

$$\texttt{wurble1 a g =}\ \boxed{\texttt{g}}\ \texttt{2 a +}\ \boxed{\texttt{g}}\ \texttt{3 a}$$

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

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

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

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

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

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

Usage type
depends on a usage context!

(*result* demand determines *argument* demands)

Two Types of Modular Program Analyses

- Forward analysis
  - “Run” the program with *abstract* input and infer the *abstract* result;
  - Examples: sign analysis, interval analysis, type checking/inference.
- Backwards analysis
  - From the expected *abstract* result of the program infer the abstract values of its inputs.

Backwards Analysis

Infers demand type basing on a context

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

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

- *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

$$C^1(U)$$

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

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

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

Each function is a
backwards demand transformer
it transforms a *context* demand to
argument demands and *fv*-demands.

How many context demands are there?

∞

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

Demand Lattice

$\top = \omega * U$

$1 * U$ $\quad$ $\omega * U(\omega * U, A)$ $\quad$ $\omega * C^{\omega}(U)$

$1 * U(A, \omega * C^{1}(U))$ $\quad$ $1 * U(\omega * U, A)$

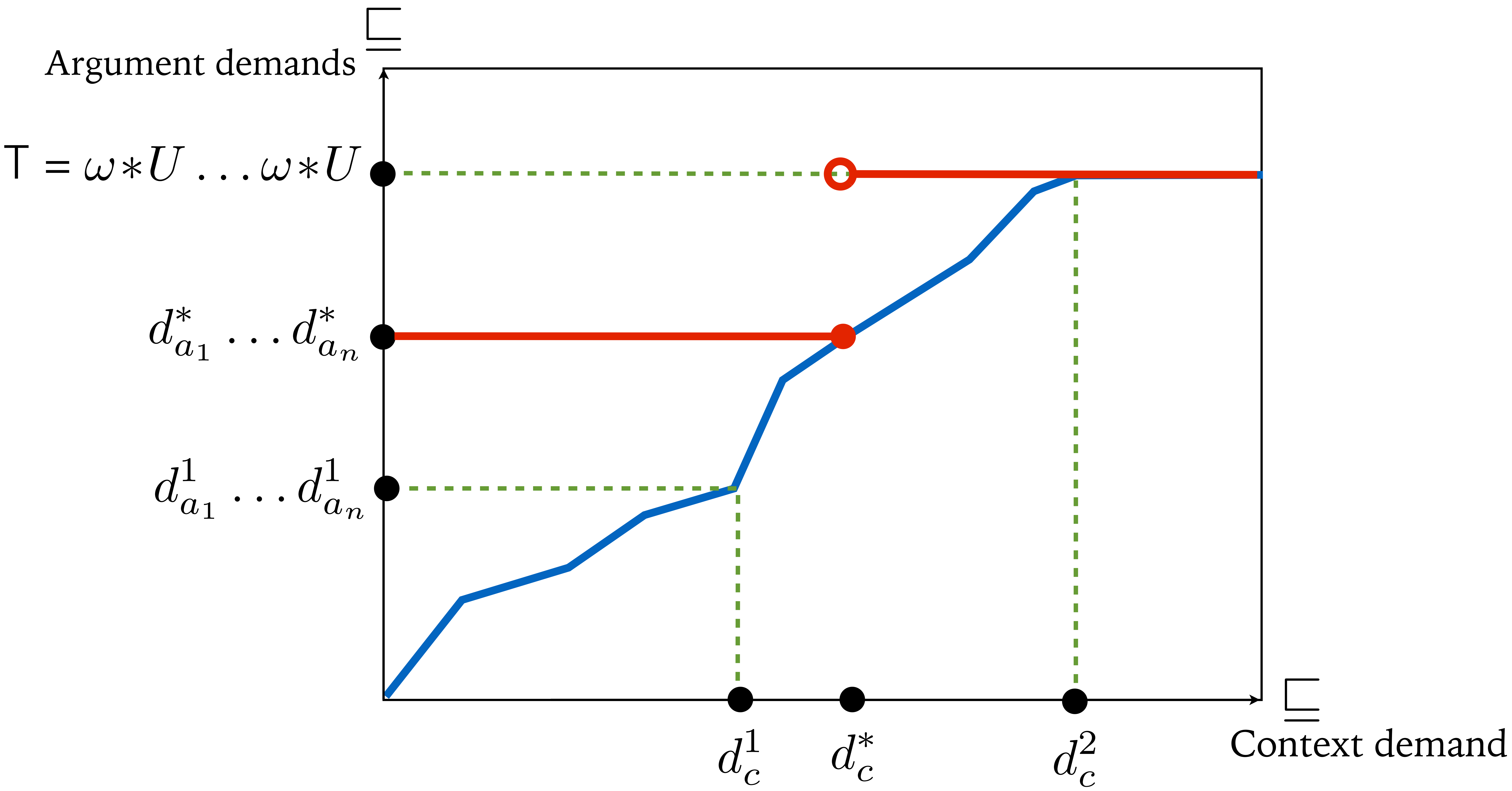
$\omega * C^{\omega}(U(A, \omega * U))$

$1 * U(A, A)$ $\quad$ $1 * C^{\omega}(U(A, \omega * U))$

$\perp = A$

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

$$P \mapsto e \downarrow d \Rightarrow \langle \tau \, ; \varphi \rangle \rightsquigarrow \mathsf{e}$$

- **let**-bindings in **e** are annotated with **m** ∈ **{0, 1, ω}** to indicate how often the let-binding is evaluated;
- Each Lambda $\boldsymbol{\lambda^{n} x . e_1}$ in **e** carries an annotation **n** ∈ **{1, ω}** to indicate how often the lambda is called.

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

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

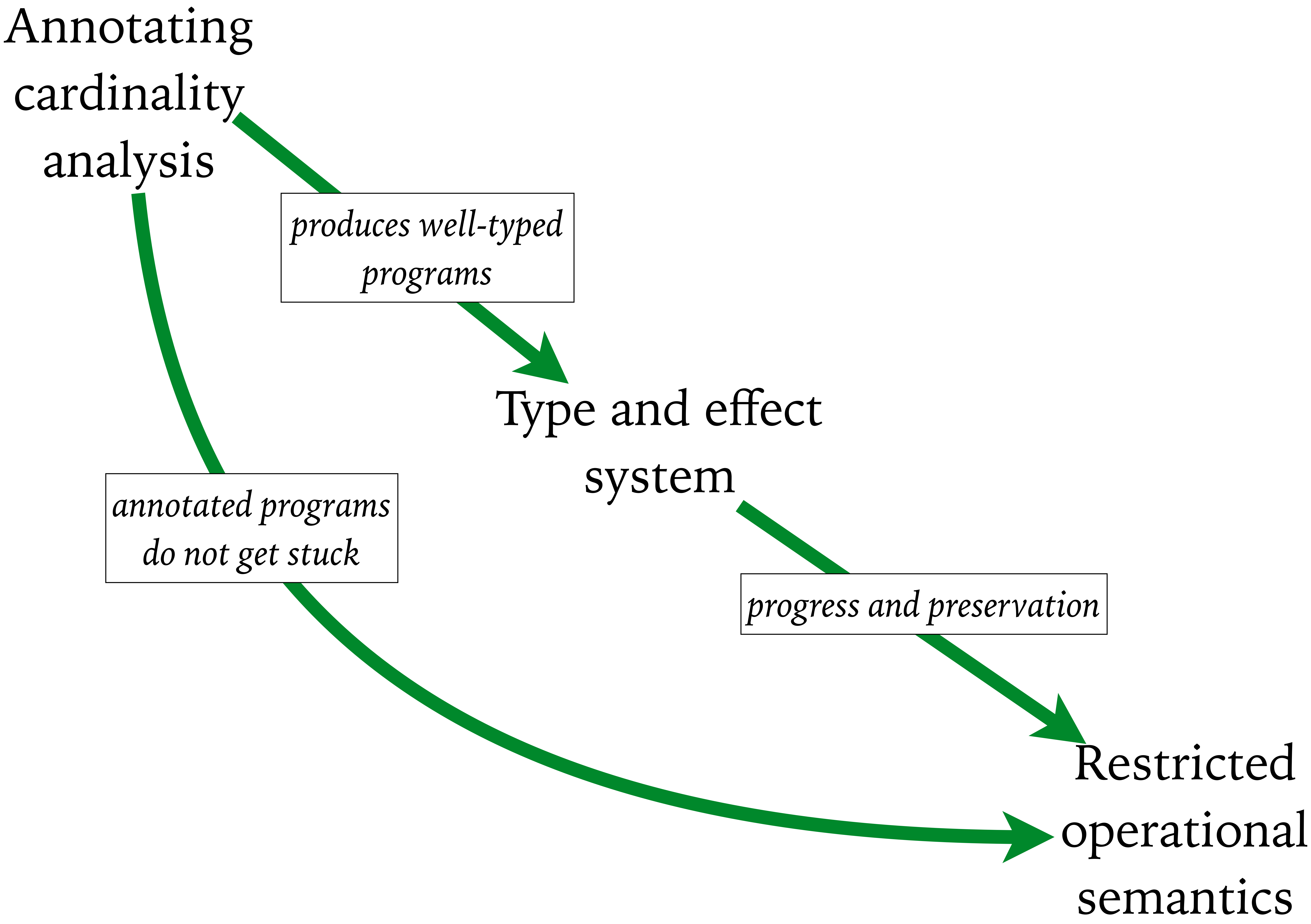
$$\rightsquigarrow$$

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

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

Small-Step CBN Machine

Sestoft:JFP97

$$e_1$$

Small-Step CBN Machine

$$\langle H_1, e_1, S_1 \rangle$$

Small-Step CBN Machine

$$\langle H_1, e_1, S_1 \rangle \longrightarrow \quad \ldots \quad \longrightarrow \langle H_n, e_n, S_n \rangle$$

$$e_1 \rightsquigarrow \mathsf{e}_1$$

Erasing Annotations

$$\mathrm{e}_1^{\natural} = e_1$$

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

Restricted CBN Machine

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

- *1*-annotated lambdas can be called at most once;
- *1*-annotated bindings can be used only once;
- *0*-annotated bindings cannot be used at all.

Soundness Theorem

An analysis-annotated program
behaves *the same way* under restricted semantics
as the original program
under the normal semantics.

Soundness Theorem

If $\epsilon \mathrel{|\!\!\blacktriangleright} e_1 \downarrow U \Rightarrow \langle \tau, \epsilon \rangle \rightsquigarrow \mathsf{e}_1$

and $\langle \epsilon \,;\, e_1 \,;\, \epsilon \rangle \longrightarrow^{k} \langle H \,;\, e_2 \,;\, S \rangle$

then

$$\langle \epsilon \,;\, \mathsf{e}_1 \,;\, \epsilon \rangle \hookrightarrow^{k}_{\exists} \langle \mathsf{H} \,;\, \mathsf{e}_2 \,;\, \mathsf{S} \rangle$$

such that $\langle \mathsf{H}^{\natural}, \mathsf{e}_2^{\natural}, \mathsf{S}^{\natural} \rangle = \langle H, e_2, S \rangle$

Cardinality-Enabled Optimisations

1. Let-in floating optimisation

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

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

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

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

Improvement Theorem 1

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

Improvement Theorem 1

For any H and S, if

$$\langle \mathsf{H} ; \texttt{let}\ z \overset{m}{=} \mathsf{e}_1\ \texttt{in}\ (\texttt{let}\ f \overset{m_1}{=} \lambda^1 x\ .\ \mathsf{e}\ \texttt{in}\ \mathsf{e}_2) ; \mathsf{S}\rangle \Downarrow^{N}$$

and $z \notin FV(\mathsf{e}_2)$

then

$$\langle \mathsf{H} ; \texttt{let}\ f \overset{m_1}{=} \lambda^1 x\ .\ (\texttt{let}\ z \overset{m}{=} \mathsf{e}_1\ \texttt{in}\ \mathsf{e})\ \texttt{in}\ \mathsf{e}_2 ; \mathsf{S}\rangle \Downarrow^{\leq N}$$

where $\sigma \Downarrow^N$ means "terminates in N steps".

2. Smart Execution

$$e_1$$

Optimised CBN Machine

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

- ***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.

Improvement Theorem 2

If $\epsilon \vdash\!\!\blacktriangleright e_1 \downarrow U \Rightarrow \langle \tau, \epsilon \rangle \rightsquigarrow \mathsf{e}_1$

and $\langle \epsilon \,;\, e_1 \,;\, \epsilon \rangle \xrightarrow{k} \boxed{\langle H \,;\, e_2 \,;\, S \rangle}$

then

$$\langle \epsilon, \mathsf{e}_1, \epsilon \rangle \stackrel{m}{\Longrightarrow} \boxed{\langle gc(\mathsf{H}), \mathsf{e}_2, gc(\mathsf{S}) \rangle}$$

such that $m \leq k$

and $\boxed{\langle \mathsf{H}^\natural, \mathsf{e}_2^\natural, \mathsf{S}^\natural \rangle} = \langle H, e_2, S \rangle$

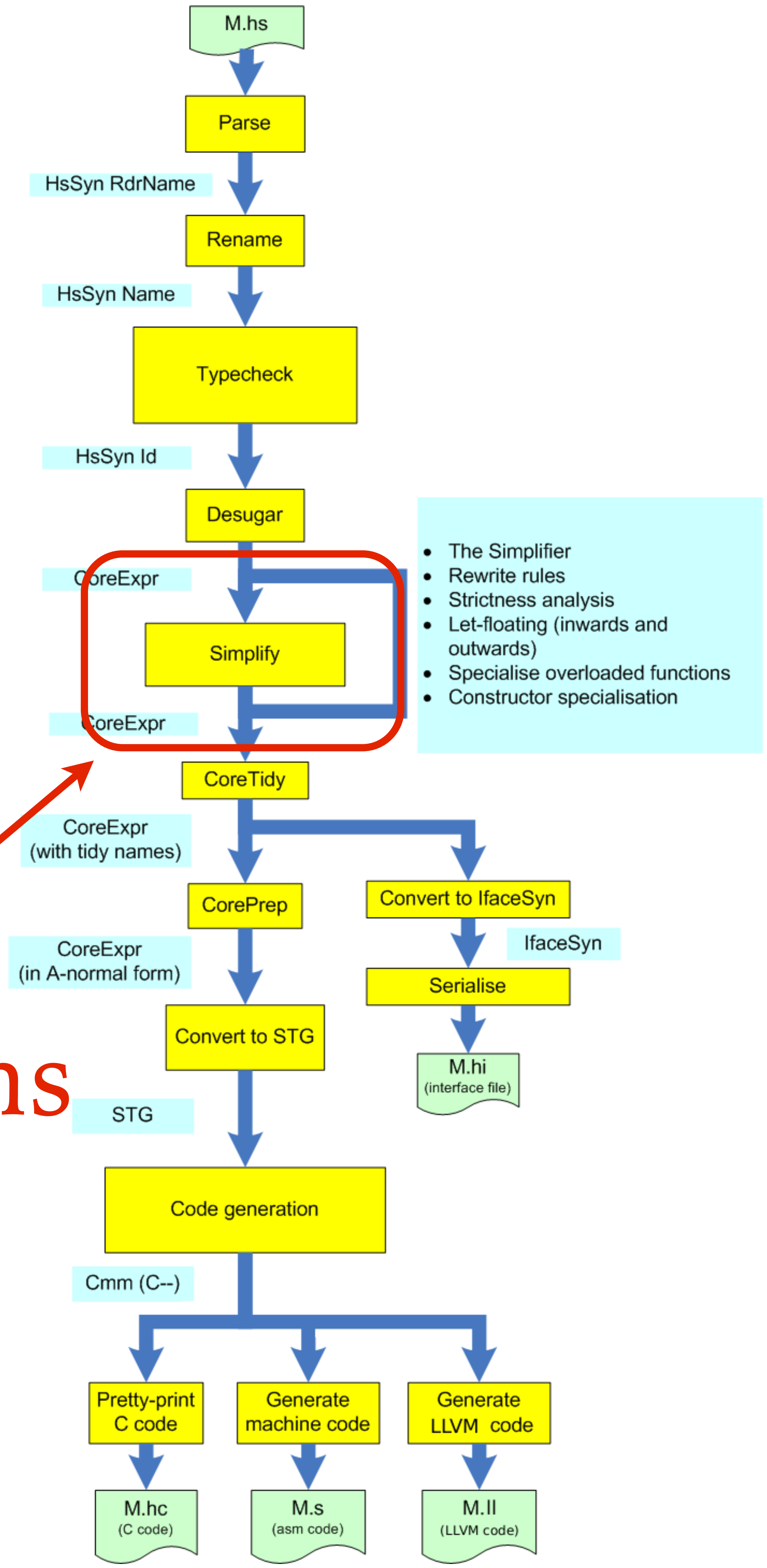
Implementation
and
Evaluation

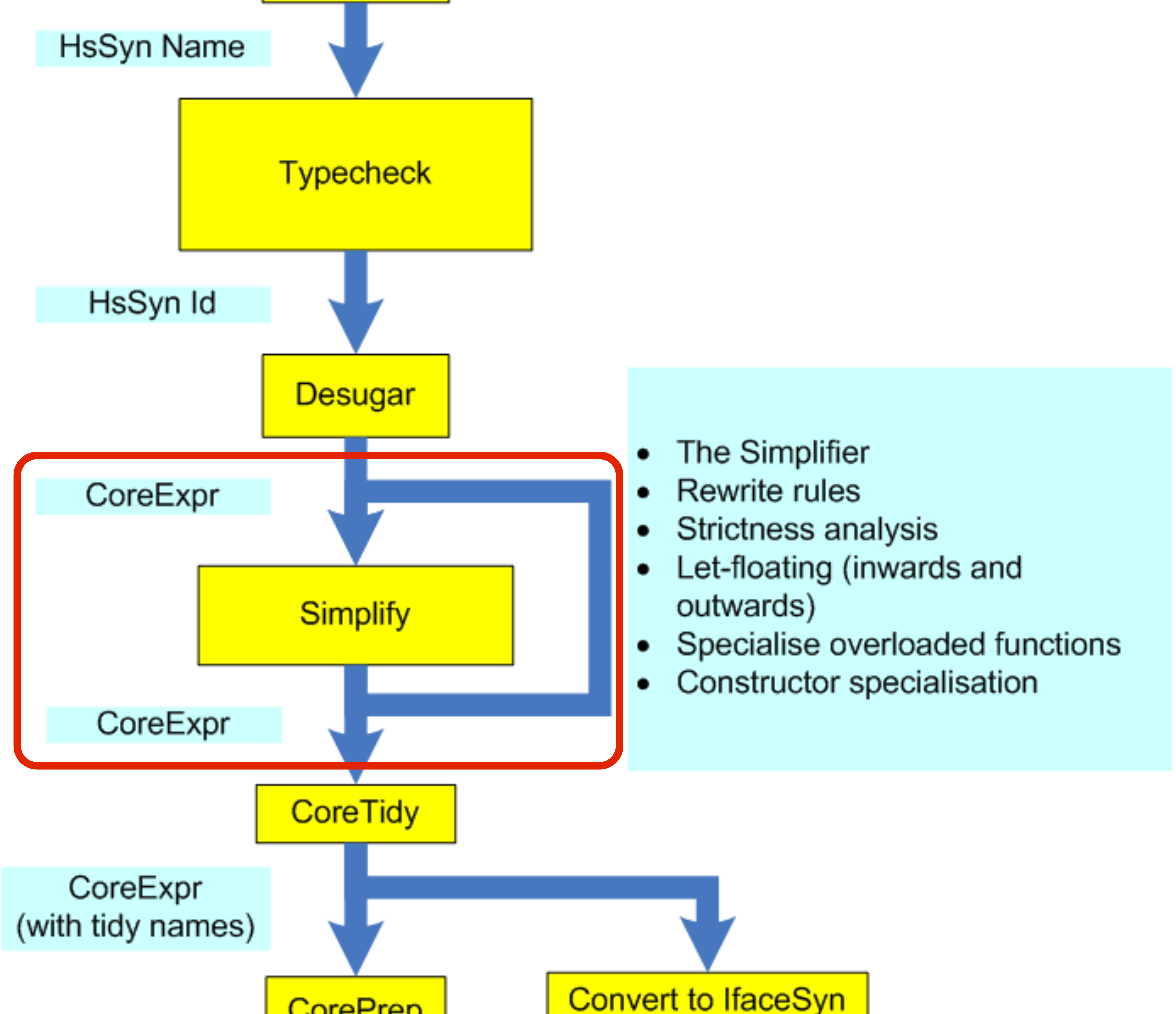
GHC Compilation Pipeline

A number of Intermediate Languages

- Haskell Source
- Core
- Spineless Tagless G-Machine
- C--
- C / Machine Code / LLVM Code

Most of interesting optimizations happen here




HsSyn Name
Typecheck
HsSyn Id
Desugar
CoreExpr
Simplify
CoreExpr
CoreTidy
CoreExpr
(with tidy names)
CorePrep
Convert to IfaceSyn
• The Simplifier
• Rewrite rules
• Strictness analysis
• Let-floating (inwards and outwards)
• Specialise overloaded functions
• Constructor specialisation

GHC Core

- A tiny language, to which Haskell sources are de-sugared;
- Based on explicitly typed System F with type equality coercions;
- Used as a base platform for analyses and optimisations;
- All names are fully-qualified;
- **if-then-else** is compiled to **case**-expressions;
- Variables have additional metadata;
- Type class constraints are compiled into record parameters.

Core Syntax

```
data Expr b
  = Var   Id
  | Lit   Literal
  | App   (Expr b) (Expr b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Tick  (Tickish Id) (Expr b)
  | Type  Type
  | Coercion Coercion

data Bind b = NonRec b (Expr b)
            | Rec [(b, (Expr b))]

type Alt b = (AltCon, [b], Expr b)

data AltCon
  = DataAlt DataCon
  | LitAlt  Literal
  | DEFAULT
```

How to See Core

Desugared Core

```
> ghc -ddump-ds Program.hs
```

Core with Strictness Annotations

```
> ghc -O2 -ddump-stranal Program.hs
```

Core after Worker/Wrapper Split

```
> ghc -O2 -ddump-worker-wrapper Program.hs
```

Try it on

```
module Program where

squash f = f 42

costly x = product [1..x]

foo xs = squash (\n -> sum (map (+ n) (map costly xs)))
```

- 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%

Beyond GHC optimisations

Program Synthesis by Type-Guided Abstraction Refinement

ZHENG GUO, UC San Diego, USA
MICHAEL JAMES, UC San Diego, USA
DAVID JUSTO, UC San Diego, USA
JIAXIAO ZHOU, UC San Diego, USA
ZITENG WANG, UC San Diego, USA
RANJIT JHALA, UC San Diego, USA
NADIA POLIKARPOVA, UC San Diego, USA

- Hoogle+ synthesis algorithm (POPL'20) relies on cardinality analysis to eliminate terms where some of the inputs are unused.

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.
- The ideas of cardinality analysis extend beyond just optimisations in GHC.

Thanks!