Improvement Theory
A Retrospective

David Sands

With thanks to former co-authors
Andy Moran and Jörgen Gustavsson

Pure Functional Programming

…is cool because there are many natural observational equivalences between programs

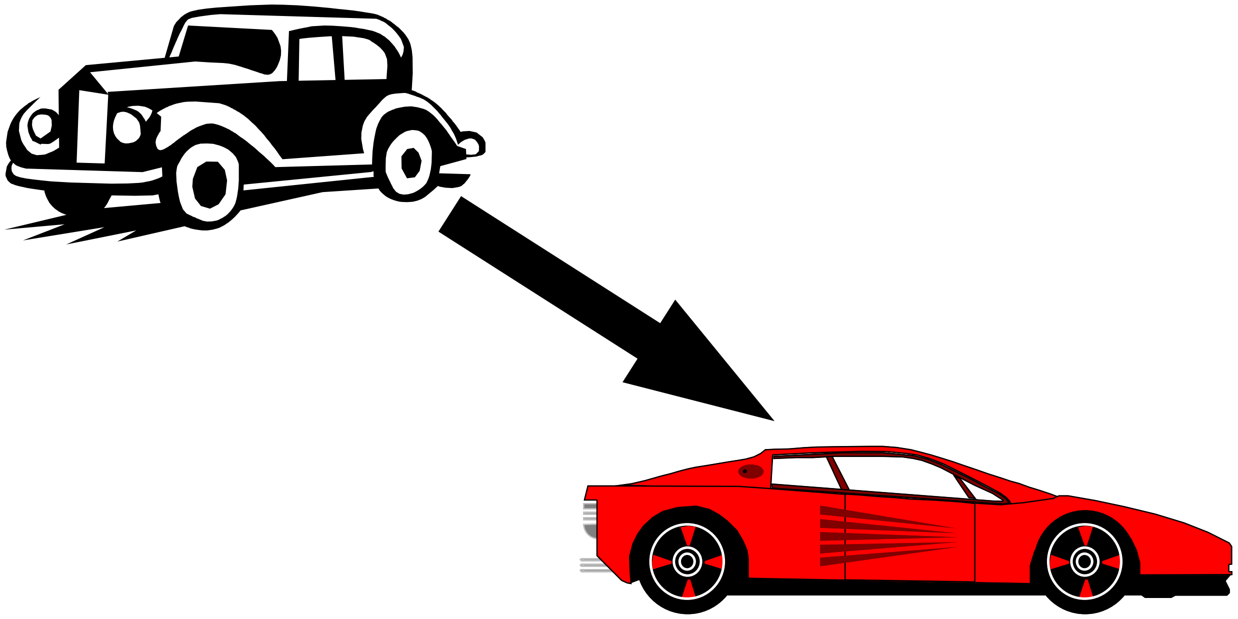
$$a + b \cong b + a$$

$$\mathrm{tail}(h{:}t) \cong t$$

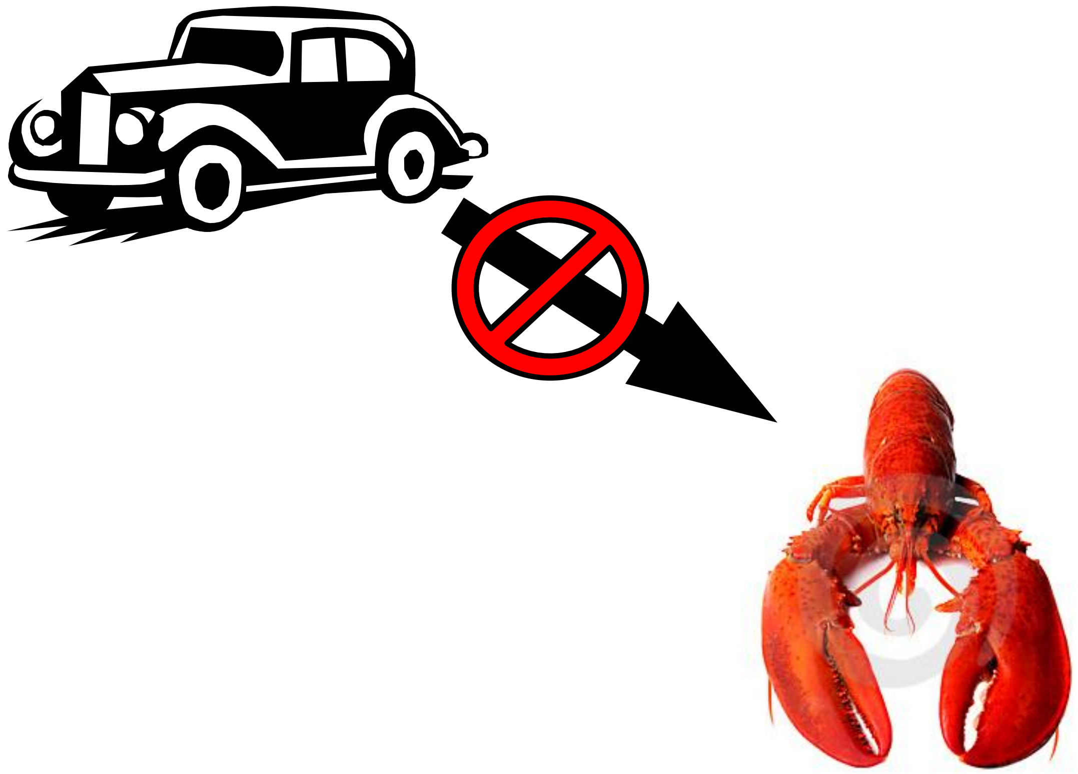
$$\mathrm{LHS} \cong \mathrm{RHS}$$

and we can use these *anywhere* in a program to obtain an equivalent program

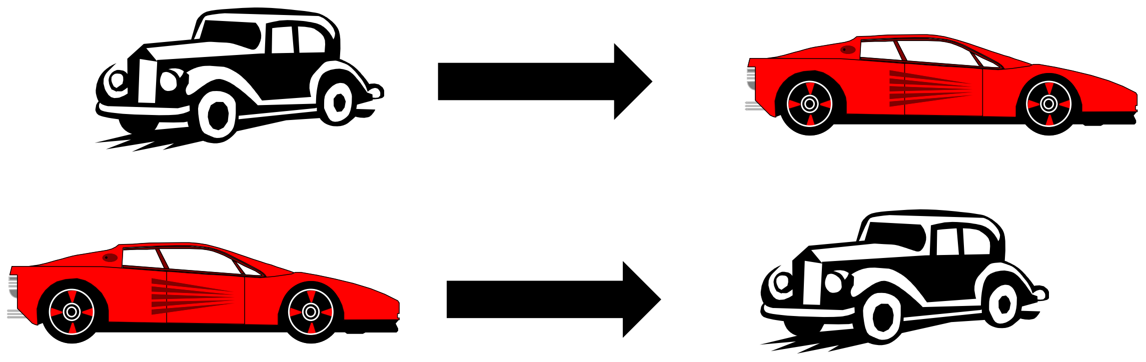
Equivalence is nice



Equivalence is nice



Problem 1



It’s all the same to equivalence

but some programs are “more equal than others”

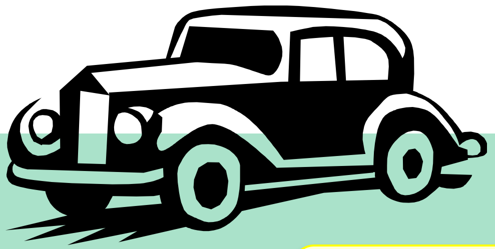
Problem 1: Example

If we replace a + b with b + a then surely nothing bad could happen… could it?

- Time complexity?
  - cannot change
- Space complexity?
  - asymptotic speedup or slowdown possible!

Problem 2

Replacing equals by equals is fool-proof
…isn't it?



```
repeat :: a -> [a]
repeat x = x : repeat x
```

t ≅ tail (h:t)

```
 RHS ≅ x : tail (x : repeat x)
     ≅ x : tail (repeat x)
```

RHS ≅ LHS

```
repeat x = x : tail (repeat x)
```

Improvement Theory

- **Improvement Theory** developed to solve problem (making sure that optimisations never make things worse, no matter what)

- Surprisingly, improvement theory provides a solution to problem

A Taster

A brief taste from a heap of papers about improvement

- Improvement Theory
- Correctness of Program Transformation by Improvement
- Reasoning about Improvement: The tick algebra
- Space Improvement

From Equivalence to Improvement

When are two programs equivalent?

Observational Equivalence

P and Q are **observationally equivalent**,

$$P \cong Q$$

iff

in whatever program context we use them, they yield the same observable outcome:

$$\forall C[.].\ \mathrm{obs}(C[P]) = \mathrm{obs}(C[Q])$$

Defined using an operational semantics

Improvement (Time)

Program phrase P is **improved by** Q,

$$P \trianglerighteq Q$$

iff

$P \cong Q$ and

in whatever program context we use them, Q is never slower than P

$$\forall C[.].\ time(C[P]) \geq time(C[Q])$$

Defined using an operational semantics

Also studied an asymptotic definition. Not in this talk

Problem 2, stated more precisely

The following proof rule is sound

$$\frac{P \cong Q}{\text{let } x = P \text{ in } x \cong \text{let } x = Q \text{ in } x}$$

It does **not** justify the bad transformation

Not true because r is a free variable

$$\frac{(\lambda\ x.\ x{:}\ r\ x) \ncong (\lambda\ x.\ x : \text{tail } (r\ x))}{\text{let } r = (\lambda\ x.\ x{:}\ r\ x) \ncong \text{let } r = (\lambda\ x.\ x : \text{tail } (r\ x))}$$

but does not allow any interesting program transformations either!

The Improvement Theorem

Sands, POPL'95
Total Correctness by Local Improvement...

In the informal transformation we used the unsound rule

$$\frac{\text{let } x = P \text{ in } P \cong \text{let } x = P \text{ in } Q}{\text{let } x = P \text{ in } x \cong \text{let } x = Q \text{ in } x}$$

The Improvement Theorem

$$\frac{\text{let } x = P \text{ in } P \rhd \text{let } x = P \text{ in } Q}{\text{let } x = P \text{ in } x \rhd \text{let } x = Q \text{ in } x}$$

The bad transformation is not justified because replacing RHS with LHS is not an improvement.

The Improvement Theorem

What's it good for?

- The first sound Unfold-Fold transformation method for functional programs [POPL'95]
- General correctness criteria for recursion-memoization-based program transformations [Sands, ESOP'95]
  - higher-order deforestation ["]
  - partial evaluation [Welinder, PhD'96]
  - supercompilation [Jonsson and Nordlander, POPL'09]
- A robust proof method insensitive to wacky language features [Lassen & Moran, MFCS'99]

The tick algebra: how reason with ⊳

Basic laws can be established (with some effort)

R[case b of {...pat$_i$ => a$_i$ ...}] ⊲⊳ case b of {...pat$_i$ => R[a$_i$] ...}

To enable equational reasoning such laws are used together with the **tick algebra**

– The tick (✓) is a representation of a basic computation step (it is just the identity function)

LHS ⊲⊳ ✓RHS

✓(if B then P else Q) ⊲⊳ if B then ✓P else ✓Q

Have we also Solved Problem 1?

- Not quite – so far we used a call-by-name computation model (easy to work with, but not resource-correct for Haskell)

Enter:

Moran & Sands, POPL’99
Improvement in a Lazy Context:
An Operational Theory for Call-by-need

- A truly semantic theory for call-by-need
  - Subsumes call-by-need lambda calculus [Ariola, Maraist, Odersky,Felliesen, Wadler, POPL’95]
  - Shows that it can only speed-up programs by a constant factor!
  - Improvement theorem etc. etc.

Space Improvement

Modest aim: when is a transformation guaranteed not to make space consumption worse by more than a constant factor $a + b \longrightarrow b + a$

```
let xs = [1..n]
    a = head xs
    b = last xs
in a + b
```

$\longrightarrow$

```
b + a
```

$\mathcal{O}(1)$ $\mathcal{O}(n)$

Space Improvement

Two major problems ('99)

- are there *any* interesting space improvements?
- even if their are, will we be able to prove them?

Key ingredients:

1. A simple abstract machine (no graph nonsense) for modelling space
2. A PhD student with a very big brain

Space Improvement

There are nontrivial space improvements!

E.g. Beta-var is a space improvement

$$(\lambda\, x.M)\ y \rhd_s M[x := y]$$

- To prove them we need to work with *non* asymptotic space improvement
- Developed fixedpoint induction principle
  - improvement theorem unsound for space
  - Example:

$$(xs +\!+\ ys) +\!+ zs \lhd\rhd_s\ xs +\!+ (ys +\!+ zs)$$

but only if heap and stack are added!

Gustavsson & Sands, ICFP'01
Possibilities and Limitations of Call-by-need Space Improvement

Never mind the ticks,
make way for the space gadgets!

- **Spikes** short-lived local maxima in heap/stack usage

  人M ≡ case True of {True => M}

- **Dummies**

  $^{\{x\}}M \equiv \text{let } y = x \text{ in } M$ (y fresh)

- **Weights**: ${}^{n}_{m}M$ long lived extra heap usage
- **Baloons**: zero weights (handle with care!)

Conclusions

- Several uses for time improvement and the improvement theorem in particular
- No takers for space improvement ☹
  - Need to make tools to make it easier to use?
- Still a lot of things that could be explored
  - Big hard problem: space-safe strictness analysis

```
f y x = x + y
```

⇒

```
f y x = y `seq` x `seq` x + y
```

IMPROVEMENT THEORY
1990 - 2001
REST IN PEACE?