The Design and Implementation of a Safe and Lightweight Haskell Compiler

Tim Chevalier

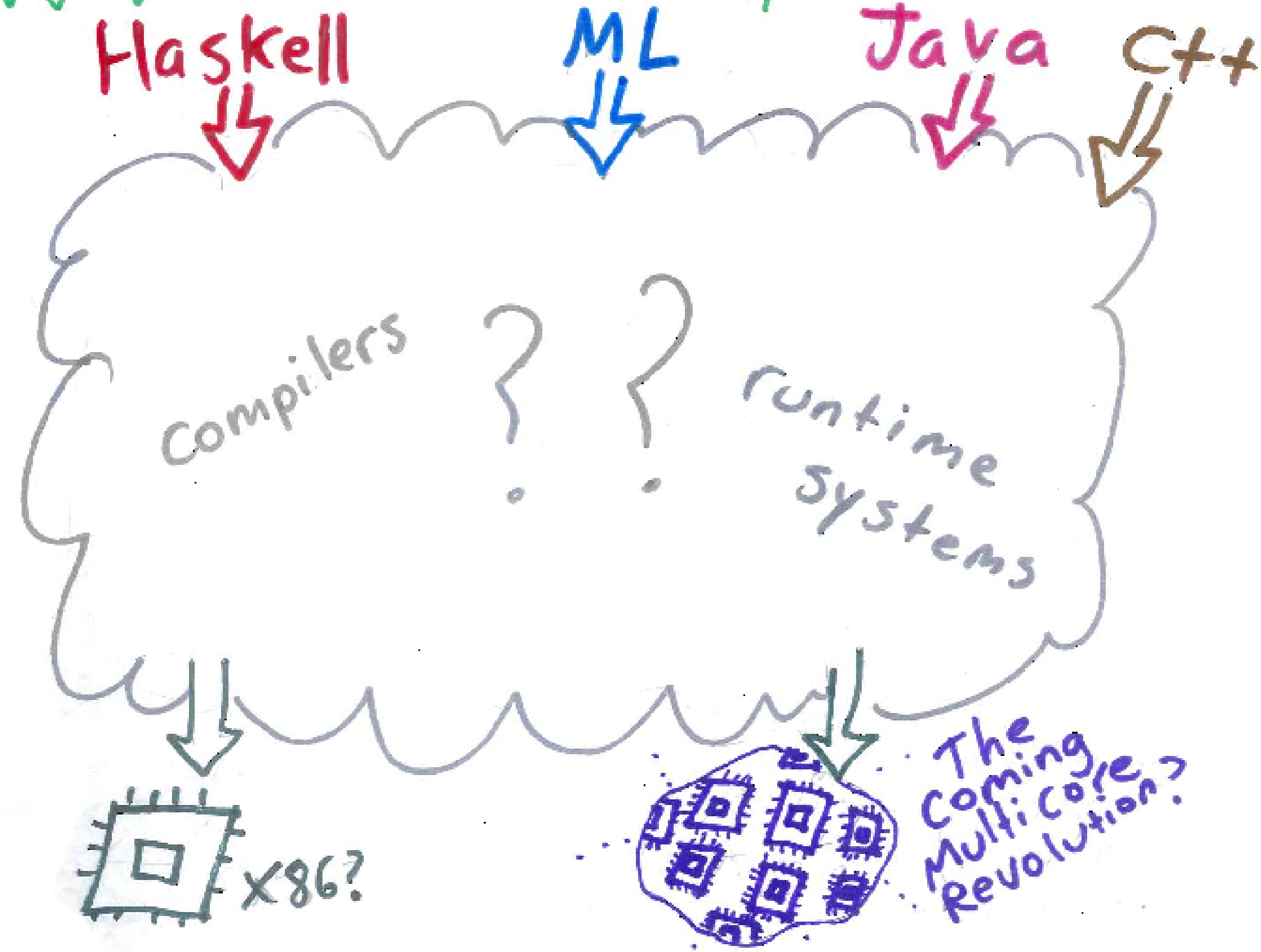
Spring 2009 RPE

Advisor: Andrew Tolmach

What do you know about your compiler?

How do you know it?

What do you Know?



The Problem

How to:

Ensure the correctness of the compiler-garbage collector interface

with a strong static guarantee

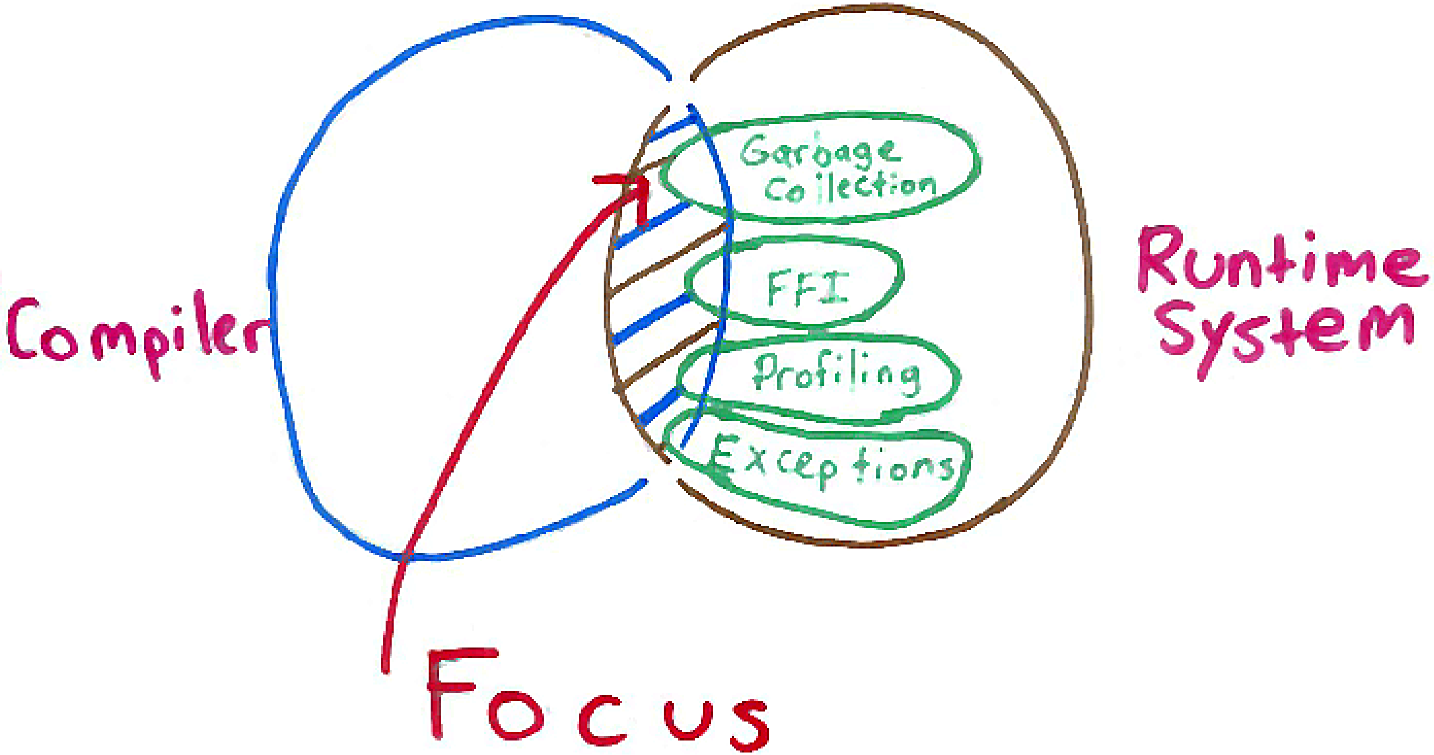
with low implementation effort

with good* performance?

My Contributions

- Designed a simple and lightweight safety-preserving compiler for Haskell
- Formulated the safety property as a simple type system
- Guaranteed safety through a combination of static and dynamic checks
- Measured performance

The Problem



Compiler
Runtime System
Garbage Collection
FFI
Profiling
Exceptions
Focus

Mozilla Foundation Security Advisory 2009-13

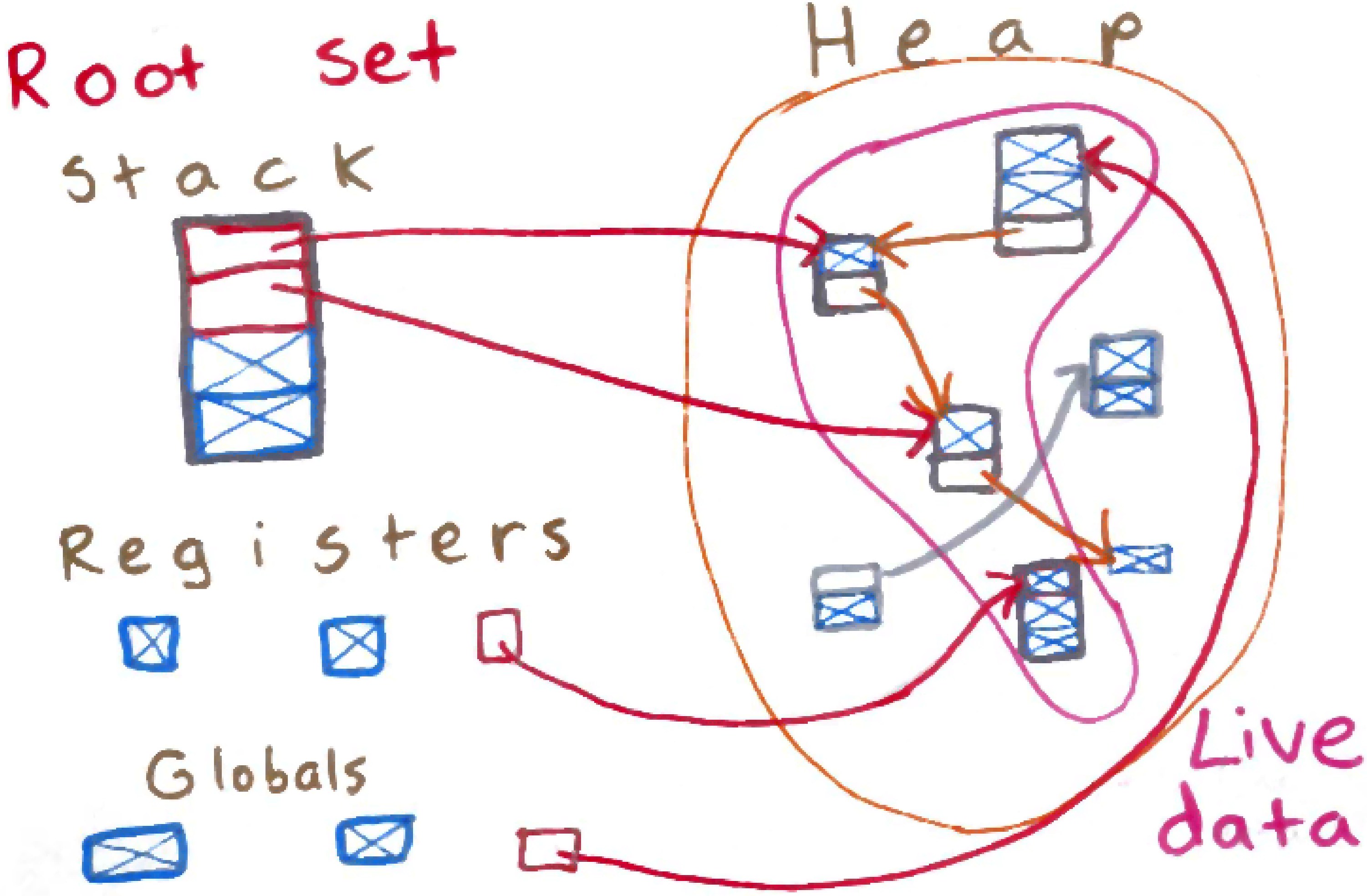
Title:	Arbitrary code execution via XUL tree element
Impact:	Critical
Announced:	March 27, 2009
Reporter:	Nils
Products:	Firefox
Fixed in:	Firefox 3.0.8

Description

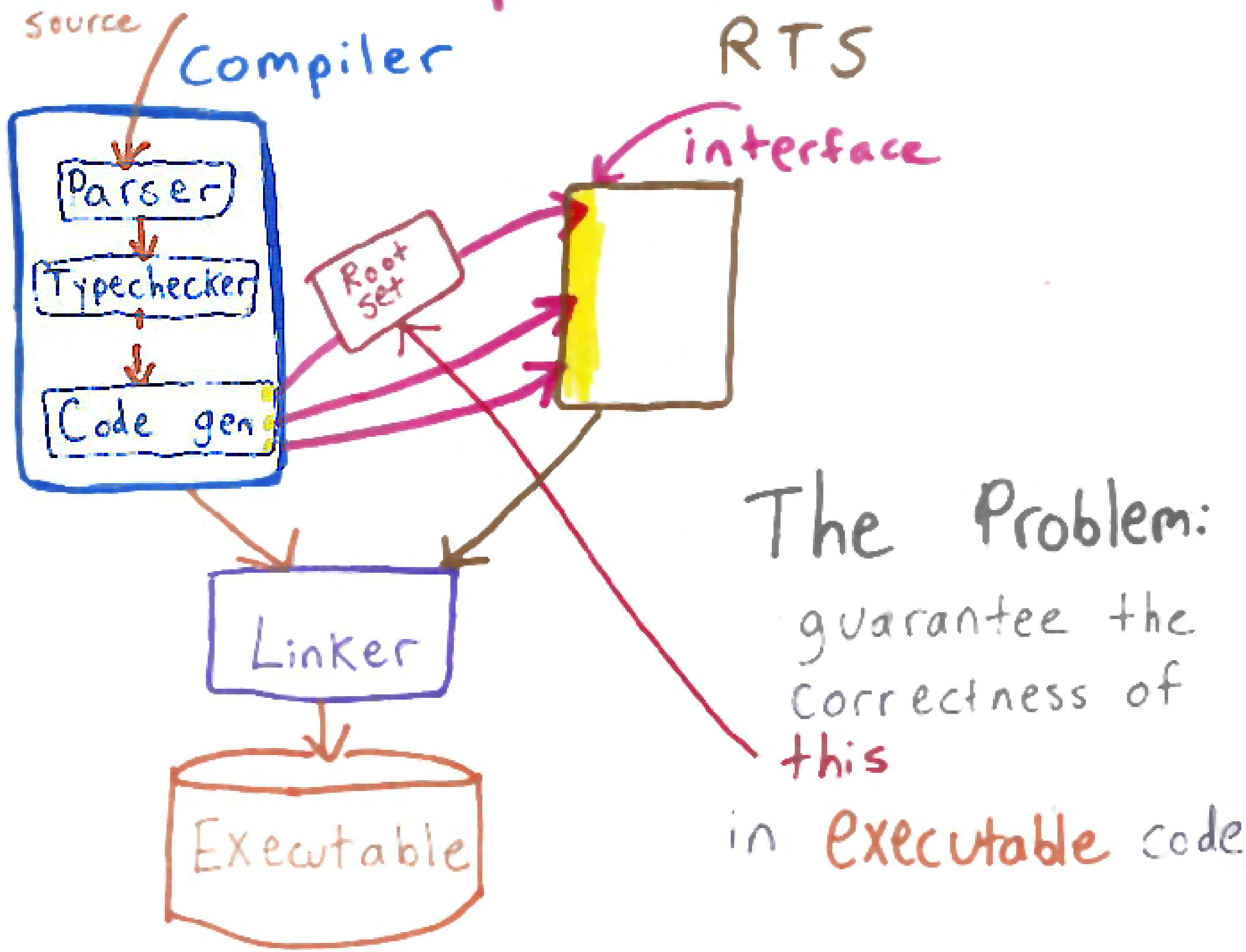
Security researcher **Nils** reported via TippingPoint's Zero Day Initiative that the XUL `tree` method `_moveToEdgeShift` was in some cases triggering garbage collection routines on objects which were still in use. In such cases, the browser would crash when attempting to access a previously destroyed object and this crash could be used by an attacker to run arbitrary code on a victim's computer.

Note: This vulnerability was used by the reporter to win the 2009 CanSecWest Pwn2Own contest.

Tracing Garbage Collection



The Compiler-GC Interface



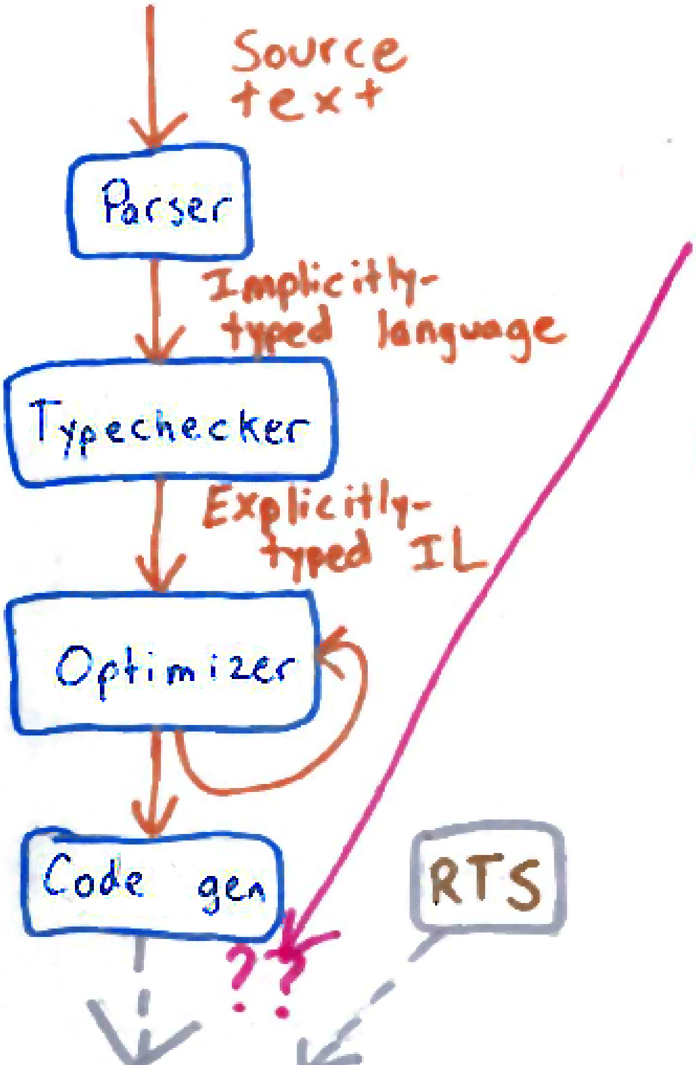
The Goal

Guarantee statically that programs pass root sets correctly

BUT

Doesn't a typed language do that already?

No!



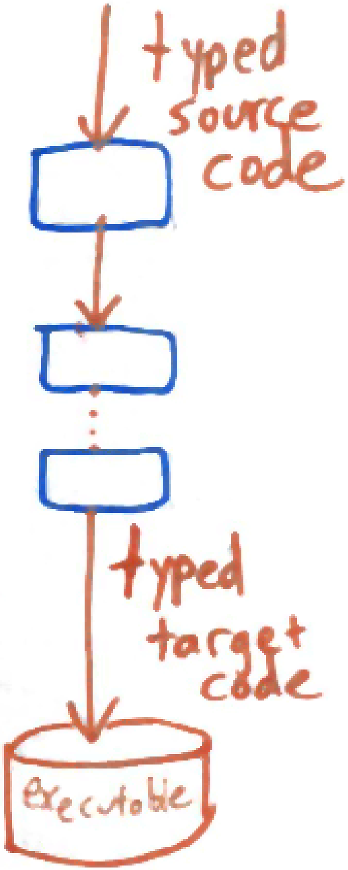
Mainstream compilers discard types at some point

Runtime system is untyped code

Source type system does not prevent compiler or RTS bugs

A stronger guarantee: type-preserving compilation

(Morrisett et al., 1999)



Prove that the compiler maps well-typed programs onto well-typed programs

Can typecheck generated code

Smaller trusted computing base

Compiler source code

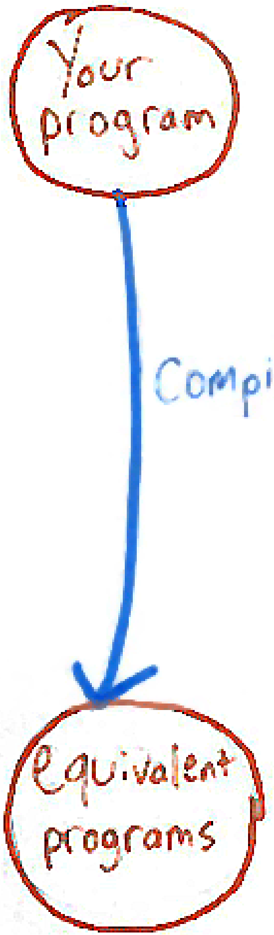
vs.

typechecker

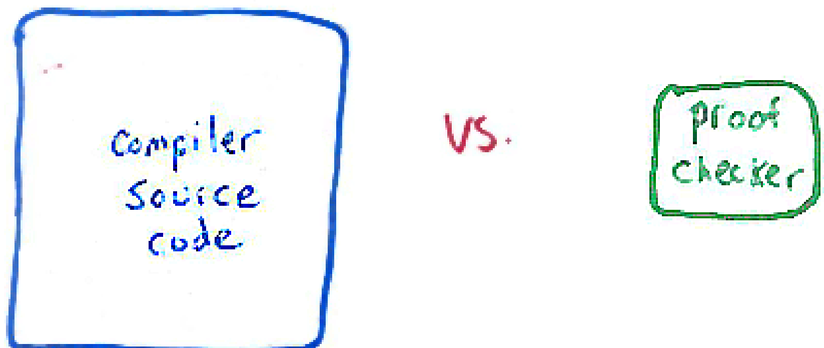
type soundness proof
type preservation proof

Still stronger guarantees: Semantics-preserving compilation

Leroy, 2006

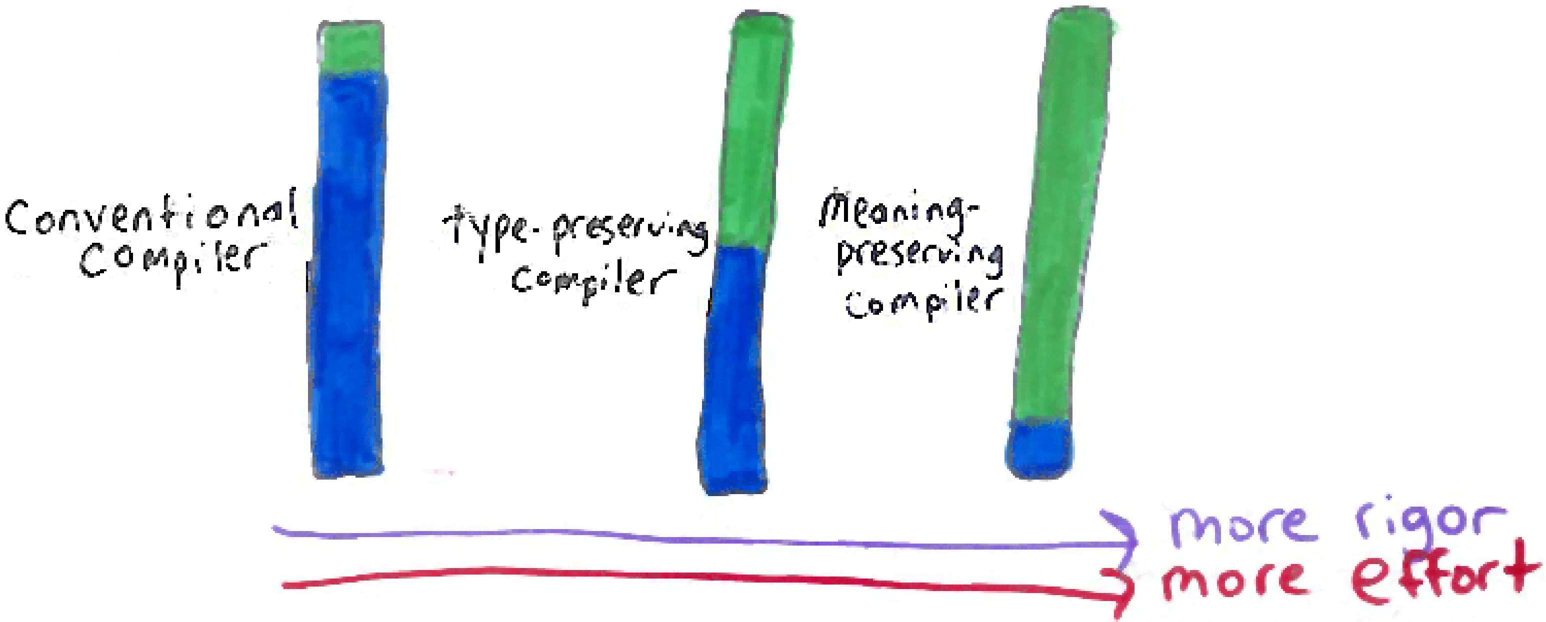


- Prove that the compiler maps any program onto one that means the same thing
- With machine-checked proof, the trusted computing base is even smaller



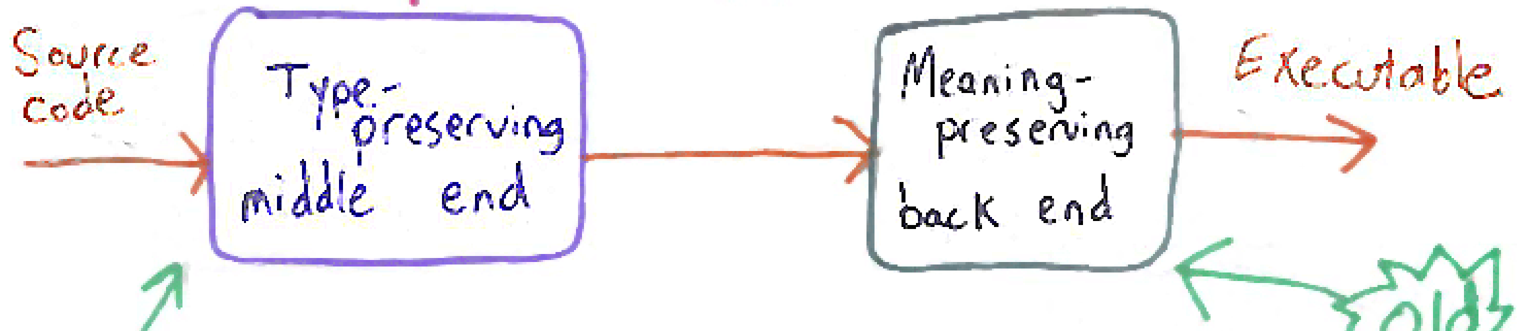
The Problem

We want to prove a property about the compiler...



... While trading off between rigor & effort

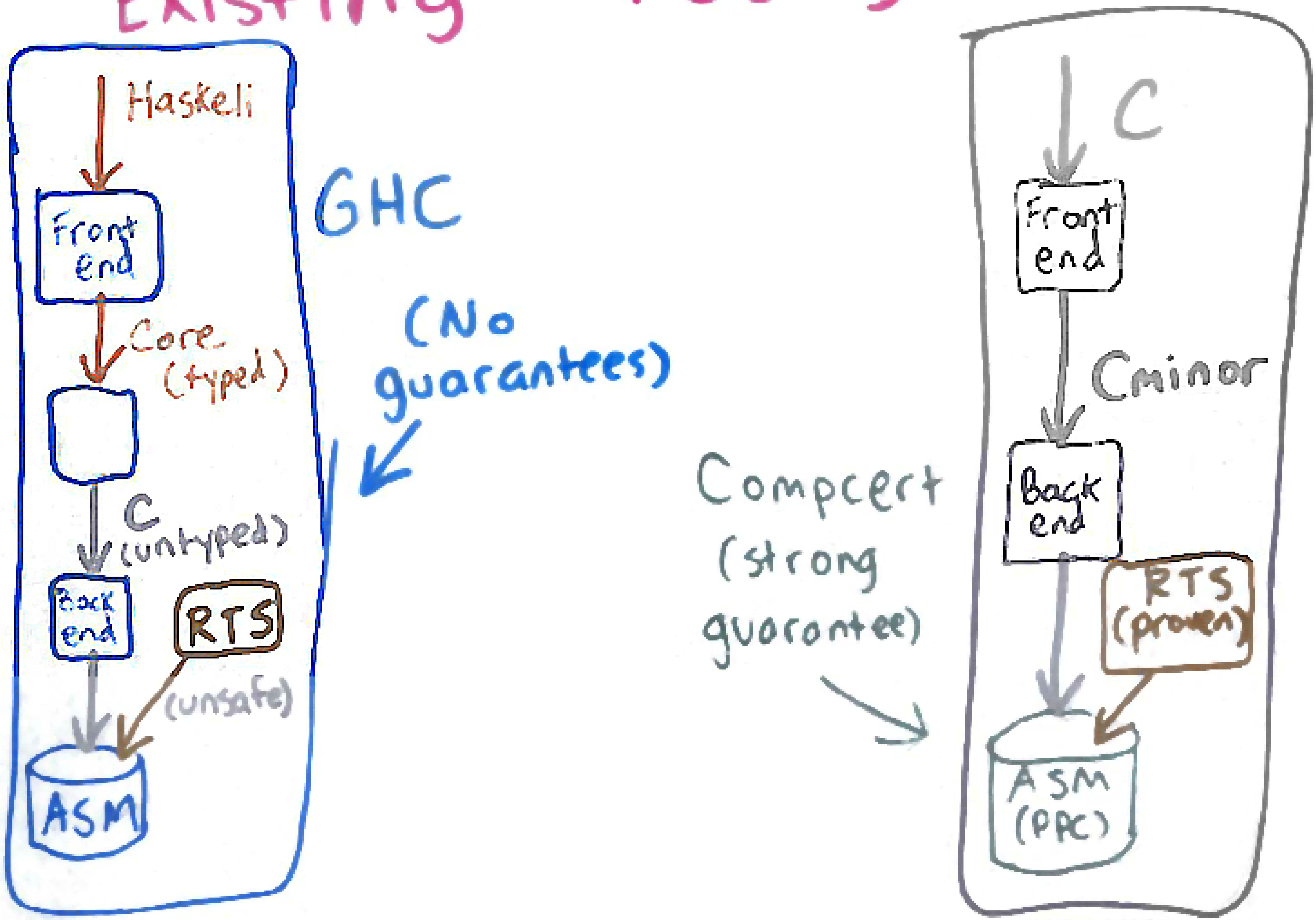
My Approach



Type preservation provides a strong guarantee that well-typed code doesn't make the GC go wrong

Meaning preservation ensures that the guarantee about source code applies to executable code

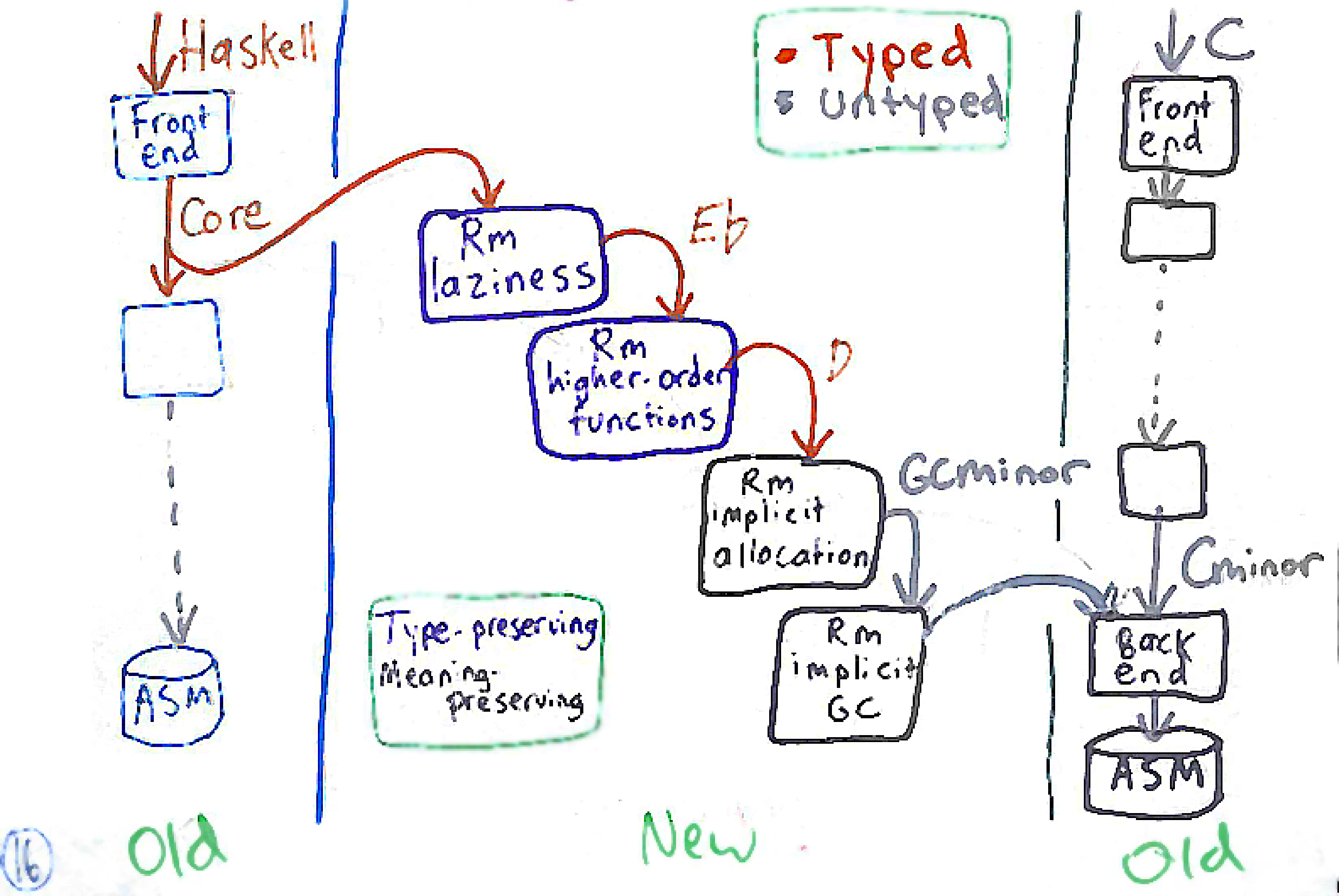
Existing Tools



Peyton Jones, 1996

Leroy, 2006

Bridging the gap



Core vs. Cminor

Core	Cminor
function types algebraic data types recursive types polymorphic types coercion types primitive types	integers floats
lazy evaluation	strict evaluation

A minimal type system

Core

Translation discards other type information

Eb

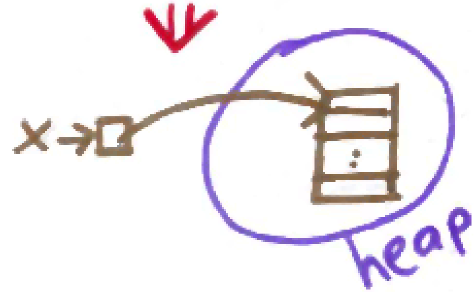
$\tau \rightarrow \mathrm{Int}$

IO

Just enough information to check the desired property

What does the type system guarantee?

$\Gamma \vdash x : \square$



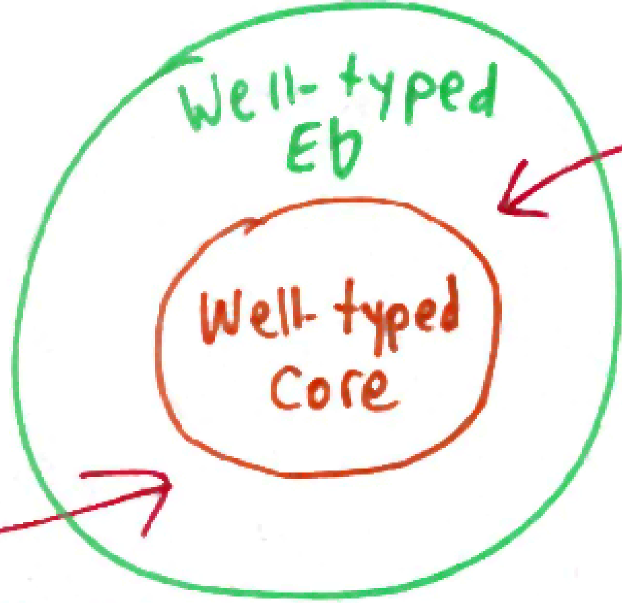
- Any variable that represents a previously allocated pointer into the heap has type $\square$
- Any other variable has type Int

What can the compiler promise?

Whenever the program calls the allocator,

- every live pointer variable is passed as a root
- no live integer variable is passed as a root

Well-typed programs



What do these programs mean?
They raise an exception.

Core's type system rejects these statically

We reject them dynamically by doing runtime checks.

Well-typed Eb code

Callee:

$$f = \lambda_{\to Int} \boxed{(g : \square)} \boxed{(x : Int)}.$$

$$\cdots\ g_{Int \to \square}\ x \cdots$$

Caller:

$$f_{\square \to Int \to Int} \underbrace{\boxed{(\lambda_{\to \square} (x : Int)\ (I\ x))}}_{\square} \underbrace{\boxed{37}}_{Int}$$

- This code typechecks (statically)
- and runs without raising an exception (dynamically)

Ill-typed Eb code

Callee:

$f = \lambda_{\to \mathrm{Int}} \boxed{(g:\square)}\ (x:\mathrm{Int}).$

$\ldots\ g_{\mathrm{Int}\to\square}\ x\ \ldots$

Caller:

$f_{\square\to\mathrm{Int}\to\mathrm{Int}}\ \underset{\mathrm{Int}}{\boxed{37}}\ (\lambda_{\to\square}(x:\mathrm{Int}).(I\ x))$

- This code fails to typecheck (statically)

Well-typed Eb code?

Callee:

$f = \lambda_{\to Int} (g : \square) (x : Int).$

$\ldots\ g_{Int \to \boxed{\square}}\ x \ldots$

Caller:

$f_{\square \to Int \to Int}\ (\lambda_{\boxed{\to Int}} (x : Int).\ 42)\ 37$

This code typechecks (statically) but raises an exception (dynamically)

The cost of checks

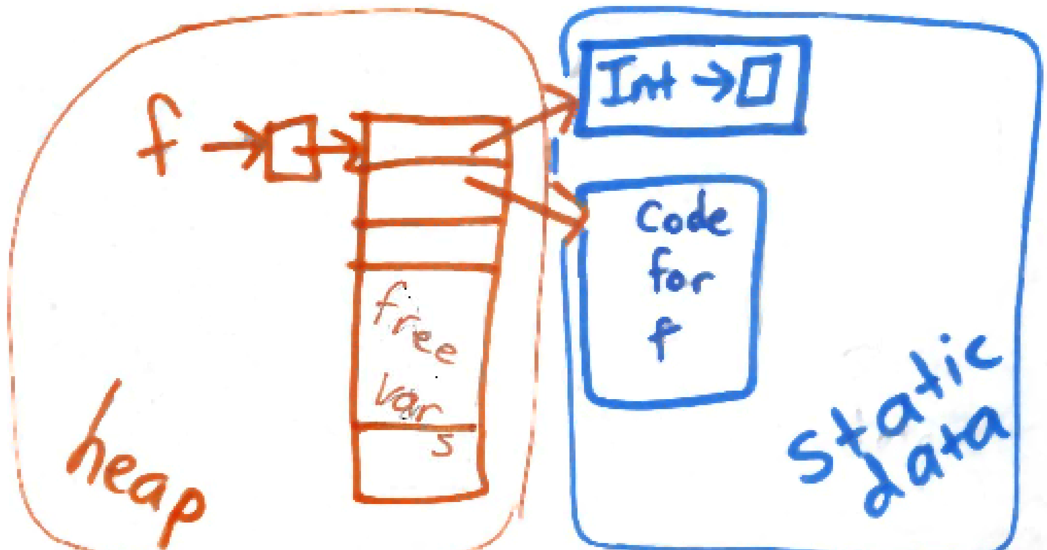
Eb

$f_{Int \rightarrow \square}$ O

Cminor

Only if checks are enabled

```
if(37 == *(*(f-4)))
  result = (*f)(0);
else
  type-error();
```



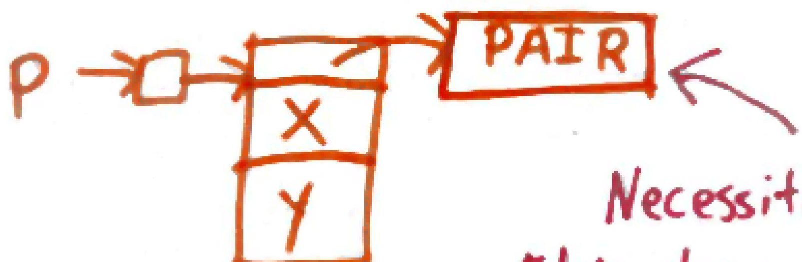
Another Kind of Check

Eb

```
case p of
  Pair (x:□) (y:□) →
    x
```

Cminor

```
if (42==*(*(p-4)))
  x=*p;
else
  type_error();
```



Necessitated by Eb's type system

Correctness properties	How we prove them
• Every variable has a consistent type	Formal static semantics for F, Eb, D Formal dynamic semantics for F, Eb, D Soundness of type systems with respect to dynamic semantics
• Translations from F → Eb, Eb → D don't change types	Type preservation of translations
• D → GCminor translation produces code that respects type distinctions (thus passing correct roots)	Formal dynamic semantics for D, GCminor Semantic preservation of translation
• GCminor → Cminor translation doesn't change this property	Semantic preservation of translation

Not yet complete

The cost of minimizing effort

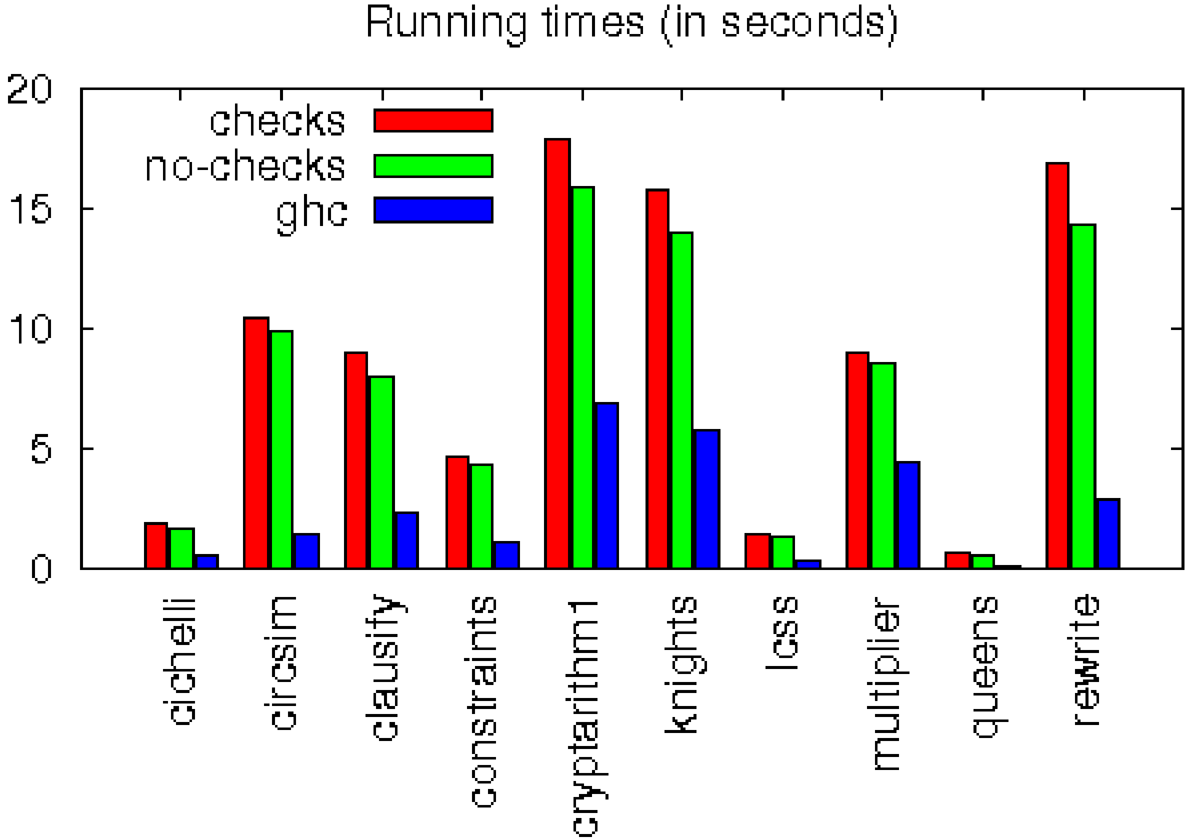
Chose ten benchmark programs from the nofib suite for Haskell

26-500 LOC

Partain, 1992

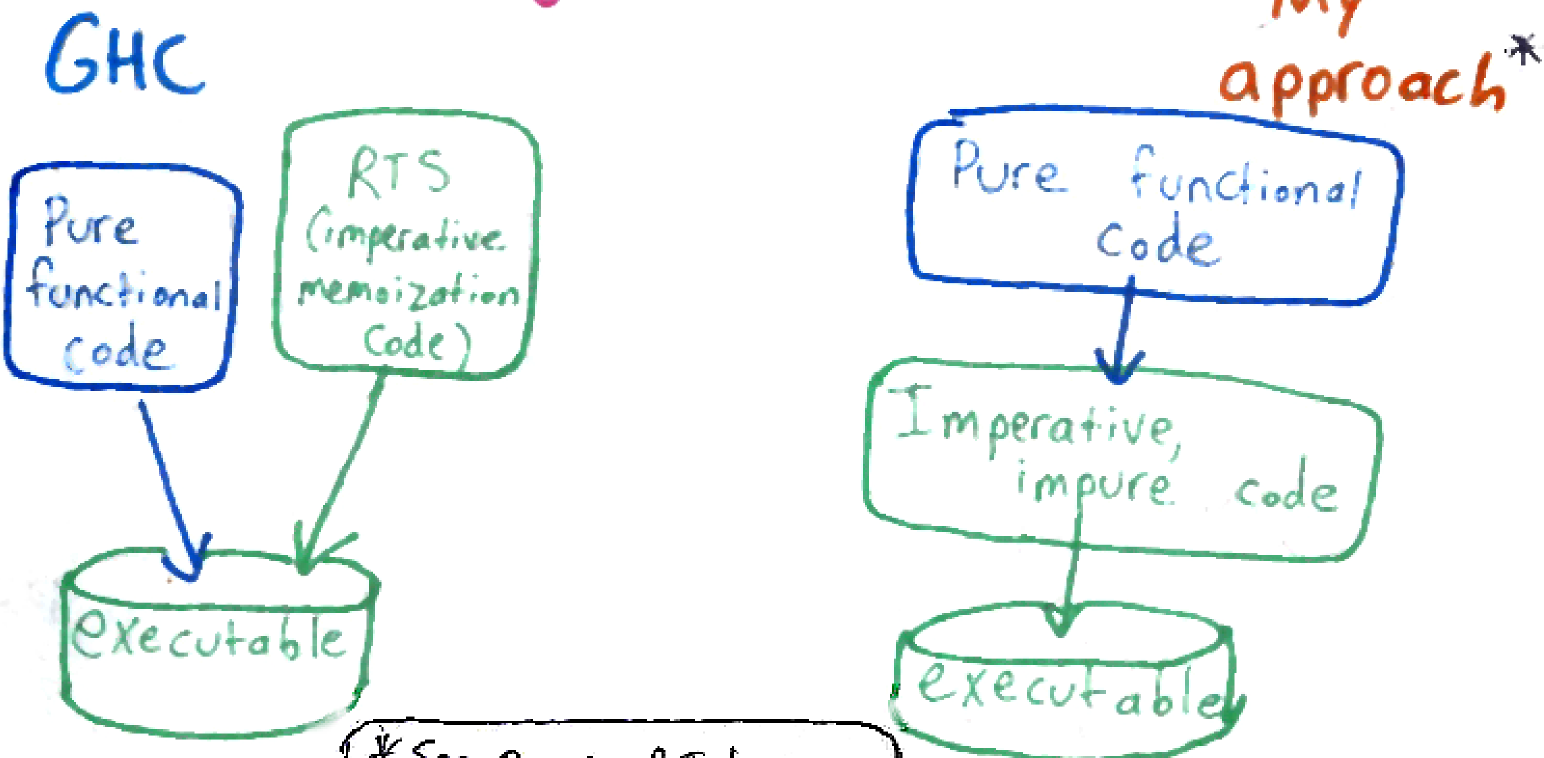
Overall running time:
avg. 4x slower than GHC

Programs with checks run 5-18% slower than programs with checks omitted



Running times (in seconds)
20
15
10
5
0
checks
no-checks
ghc
cichelli
circsim
clausify
constraints
cryptarithm1
knights
lcss
multiplier
queens
rewrite

A source of overhead: Compiling laziness



* See: Boquist & Johnsson, 1996
Faxén, 1997

My Contributions

- Showed that, with low implementation effort, a compiler can give a strong static guarantee that code uses the GC correctly.

- Measured the cost of providing the guarantee through a combination of static and dynamic checks.

Conclusions

- More work is needed to determine how much overhead is inherent to the task of increasing safety, and how much is due to naïve implementation.
- My results provide a preliminary suggestion that increasing confidence costs no more than disabling optimization.

Thanks to:

Ki Yung Ahn · Iavor Diatchki · Akshay Dua · Rafael J. Fernández-Moctezuma · Tom Harke · Jim Hook · Phil Howard · Mark Jones · Rashawn Knapp · Chuan-Kai Lin · Ralph London · Andrew McCreight · Phillip Sitbon · Andrew Tolmach

Paper: http://cs.pdx.edu/~tjc/tjc-rpe.pdf