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?

Haskell  ML  Java  C++

Compilers  ??  runtime systems

The Coming Multicore Revolution?
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

Focus

Runtime System

Garbage Collection

FFI

Profiling

Exceptions
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

Root set
- Stack
- Registers
- Globals

Heap

Live data
The Compiler-GC Interface

The Problem: guarantee the correctness of this in executable code
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

Can typecheck well-typed programs onto well-typed generated code.

Prove that the compiler maps typed code to typed code.

Smaller trusted computing base.

Type-soundness proof.

Type preservation proof.

Maris et al., 1999.
Still stronger guarantees: Semantics-preserving Compilation

- 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...

Conventional Compiler  type-preserving compiler  Meaning-preserving compiler

more rigor  more effort

...while trading off between rigor & effort
My Approach

Source code → Type-preserving middle end → Meaning-preserving back end → Executable

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

Haskell

Front End

Core (typed)

Back End

C (untyped)

RTS

 UNSAFE

ASM

GHC

C

(No guarantees)

Cminor

Front End

Back End

RTS (proven)

ASM (PPC)

CompCert (strong guarantee)

Peyton Jones, 1996

Leroy, 2006
Bridging the gap

- Typed
- Untyped

Haskell

Front end

Core

Rm laziness

Rm higher-order functions

Eb

Type-preserving
Meaning-preserving

Rm implicit allocation

GCminor

Rm implicit GC

Cminor

C

Front end

Back end

ASM

Old

New

Old
Core vs. C minor

<table>
<thead>
<tr>
<th>Core</th>
<th>C minor</th>
</tr>
</thead>
<tbody>
<tr>
<td>function types</td>
<td>integers</td>
</tr>
<tr>
<td>algebraic data types</td>
<td></td>
</tr>
<tr>
<td>recursive types</td>
<td>floats</td>
</tr>
<tr>
<td>polymorphic types</td>
<td></td>
</tr>
<tr>
<td>coercion types</td>
<td></td>
</tr>
<tr>
<td>primitive types</td>
<td></td>
</tr>
<tr>
<td>lazy evaluation</td>
<td>strict evaluation</td>
</tr>
</tbody>
</table>
A minimal type system

Core

Translation discards other type information

Just enough information to check the desired property
What does the type system guarantee?

- Any variable that represents a previously allocated pointer into the heap has type `\texttt{_ptr}
- Any other variable has type `\texttt{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

Well-typed Core

Core's type system rejects these statically.

Well-typed Eb

What do these programs mean? They raise an exception.

We reject them dynamically by doing runtime checks.
Well-typed Eb code

**Callees:**

\[ f = \lambda \rightarrow \text{Int} \ (g : \square) (x : \text{Int}) \]

\[ \ldots \ g_{\text{Int} \rightarrow \square} \times \ldots \]

**Caller:**

\[ f_{\square \rightarrow \text{Int} \rightarrow \text{Int}} \ (\lambda_{\rightarrow \square} (x : \text{Int}) (I \times)) \]

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

Callee:
\[ f = \lambda \text{Int} \rightarrow (g: \square) (x: \text{Int}). \]
\[ \ldots g: \text{Int} \rightarrow \square x \ldots \]

Caller:
\[ f: \square \rightarrow \text{Int} \rightarrow \text{Int} \quad 37 \quad (\lambda_{\square} (x: \text{Int}). (1x)) \]
\[ \text{Int} \]

This code fails to typecheck (statically)
Well-typed Eb code?

Callee:

\[ f = \lambda \rightarrow \text{Int} \ (g:\Box) \ (x:\text{Int}). \]

\[ \ldots \ g_{\text{Int}} \Box \times \ldots \]

Caller:

\[ f_{\Box} \rightarrow \text{Int} \rightarrow \text{Int} \ (\lambda \rightarrow \text{Int} \ (x:\text{Int}). \ 42) \ 37 \]

This code typechecks (statically) but raises an exception (dynamically)
The cost of checks

\[ E_b \]

\[ f \_{\text{Int}} \rightarrow 0 \ 0 \]

Cminor

```c
if (37 == *(*(f-4)))
  result = (*f)(0);
else
  type_error();
```

Only if checks are enabled

![Diagram showing memory layout with heap, free vars, code for f, and static data]
Another kind of check

E_b

Case P of pair (x;y) (y;b) →

p

Necessitated by E_b's type system

else

Cminor

\[ x = *p \]

\[ (x(yd))^* = 2h(t) \]
Correctness properties

- Every variable has a consistent type
- Translations from F → Eb, Eb → D don't change types

How we prove them

- Formal static semantics for F, Eb, D
- Formal dynamic semantics for F, Eb, D
- Soundness of type systems with respect to dynamic semantics
- Type preservation of translations

- D → Gcminor translation produces code that respects type distinctions (thus passing correct roots)
- Gcminor → Cminor translation doesn't change this property

Not yet complete
The cost of minimizing effort

Chose ten benchmark programs from thenofib suite for Haskell
26-500 Loc

Overall running time:
  avg. 4x slower than GHC

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

Partain, 1992
A source of overhead: Compiling laziness

GHC

Pure functional code

RTS (imperative memoization code)

Executable

My approach

Pure functional code

Imperative, impure code

Executable

* 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 Ova.
Rafael J. Fernández-Moctezuma. Tom Harke.
Rashawn Knapp. Chuan-Kai Lin.
Ralph London. Andrew McCready.
Phillip Sitbon. Andrew Tolmach.