Closure Conversion is Safe for Space

Zoe Paraskevopoulou Andrew W. Appel

Princeton University

ICFP'19
Berlin, Germany

Compiler correctness

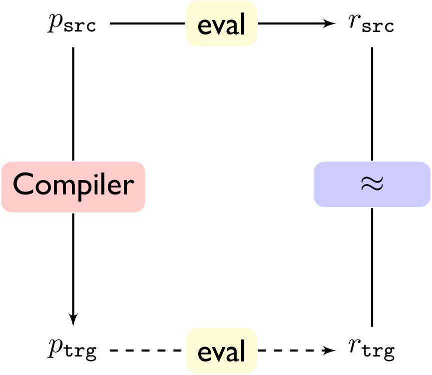
CompCert

CAKEML
A Verified Implementation of ML

Pilsner

Vellvm
verified
LLVM

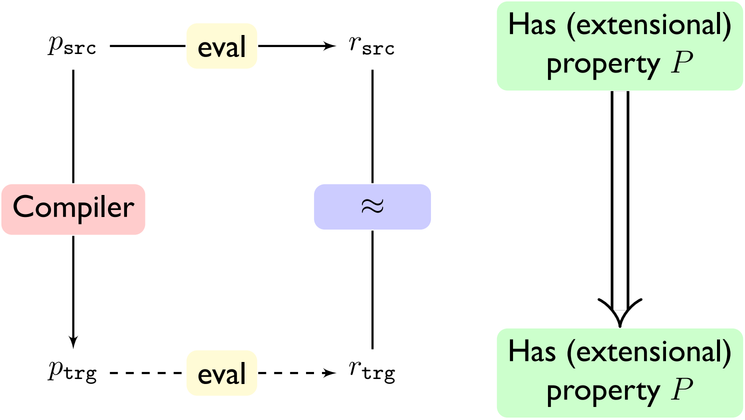
CertiCoq



Compiler correctness

CompCert

Pilsner



Compiler correctness

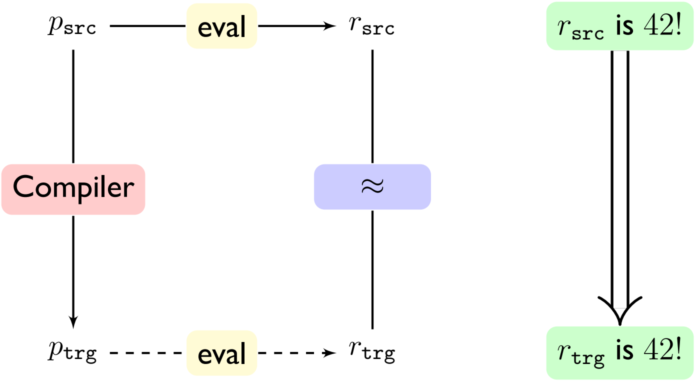
CompCert

CAKEML
A Verified Implementation of ML

Pilsner

Vellvm
verified
LLVM

Certi Coq



Compiler correctness

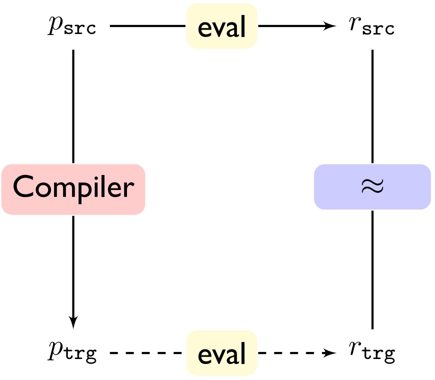
CompCert

CAKEML
A Verified Implementation of ML

Pilsner

Vellvm
verified
LLVM

CertiCoq



$p_{\mathtt{src}}$ runs in T time and M space

Preservation of Resource Consumption

Compiler transformations may leak resources!

Grokking V8 closures for fun (and profit?)

Vyacheslav Egorov on 23 sep 2012

I was thinking about writing a smallish blog post summarizing my thoughts on closure
instance field performance as a reply to Marijn Haverbeke's post which postulates in
when I realized that this is an ideal candidate for a longer post that illustrates how
closures and how these design decisions affect performance.

Contexts

function carries arou
s when you execute fun
thing:

variables x and y becau
ists.

tes an object called C

10 January 2016

Beware of the closure memory leak in Javascrip

I discussed closures in javascript in a previous article and the impact hit that you c
if you use them too much; remember that in js, each function is an object, so the c
has a cost.
This article focuses on some examples of memory leaks using closures.

How to create a memory leak in a Javascript -browser, node.js-

The major point to remember is that in a javascript closure
same context.

```
var res;

function outer() {
        var largeData = new Array(10000000);
        var oldRes = res;

    /* Unused but leaks? */
        function inner() {
                if (oldRes) return largeData;
        }

        return function(){};
}

setInterval(function() {
        res = outer();
}, 10);
```

METEOR Sign in

An interesting kind of JavaScript memory leak

David Glasser Follow
Aug 12, 2013 · 6 min read

Recently, Avi and David tracked down a surprising JavaScript memory leak in Meteor's live HTML template rendering system. The fix will be in the 0.6.5 release (in its final stages of QA right now).

I searched the web for variations on `javascript closure memory leak` and came up with nothing relevant, so it seemed like this is a relatively little-known issue in the JavaScript context. (Most of what you find for that query

Preservation of Resource Consumption

Extend compiler correctness statement to include preservation of *time* and *space* consumption

- extensions of CompCert[1] [2]
- Tricky for higher-order, memory-managed languages

---

[1] F. Besson, S. Blazy and P. Wilke. *A Memory-Aware Verified C Compiler Using Pointer as Integer Semantics*, ITP'2017.

[2] Q. Carbonneaux, J. Hoffman and T. Ramananandro, *End-to-end verification of stack-space bounds for C programs*, PLDI'14.

This work

First formal proof that closure conversion is *safe for space*

- Implemented and verified in Coq for CertiCoq
- Profiling semantics for source and target
- Logical relation framework

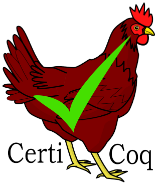
NEW Principled way of reasoning about intensional properties

NEW Divergence preservation (w/ memory consumption)

CertiCoq

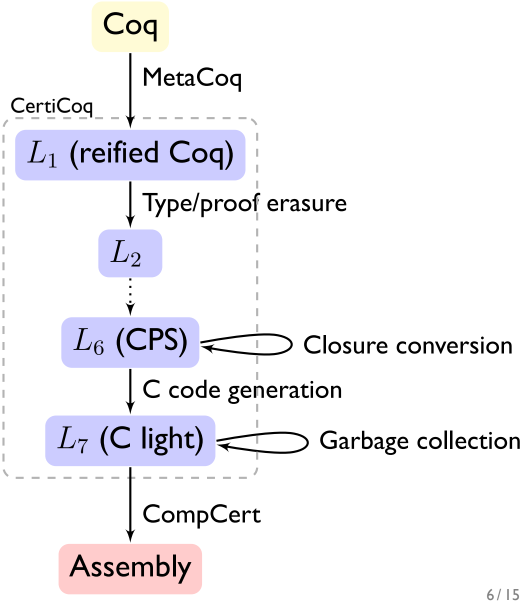
A verified compiler for Gallina.

Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver



- Implemented and verified in Coq.
- GC implemented in C, verified in Coq (VST).

Shengyi Wang, Qinxiang Cao, Anshuman Mohan, and Aquinas Hobor



Closure Conversion

Eliminates free variables by explicitly constructing a closure environment upon function definition, and passing it as an argument at call sites.

$$\texttt{let}\ f\ x = x + y + z\ \texttt{in}$$
$$\ldots$$
$$f\ 3$$

Closure Conversion

Eliminates free variables by explicitly constructing a closure environment upon function definition, and passing it as an argument at call sites.

$$\texttt{let}\ f\ x = x + y + z\ \texttt{in}$$
$$\ldots$$
$$f\ 3$$

$$\leadsto$$

$$\texttt{let}\ f_{\texttt{env}} = (y, z)\ \texttt{in}$$
$$\texttt{let}\ f_{\texttt{code}}\ x\ env\ = x + env\,\texttt{.0} + env\,\texttt{.1}\ \texttt{in}$$
$$\texttt{let}\ f = (f_{\texttt{code}}, f_{\texttt{env}})\ \texttt{in}$$
$$\ldots$$
$$\texttt{let}\ f_{\texttt{code}}\ = f\,\texttt{.0}\ \texttt{in}$$
$$\texttt{let}\ f_{\texttt{env}}\ = f\,\texttt{.1}\ \texttt{in}$$
$$f_{\texttt{code}}\ f_{\texttt{env}}\ 3$$

Closure Conversion

Eliminates free variables by explicitly constructing a closure environment upon function definition, and passing it as an argument at call sites.

$$\texttt{let}\ f\ x = x + y + z\ \texttt{in}$$
$$\ldots$$
$$f\ 3$$

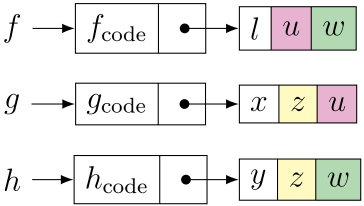
$$\rightsquigarrow$$

$$\texttt{let}\ f_{\texttt{env}} = (y, z)\ \texttt{in}$$
$$\texttt{let}\ f_{\texttt{code}}\ x\ env = x + env\texttt{.0} + env\texttt{.1}\ \texttt{in}$$
$$\texttt{let}\ f = (f_{\texttt{code}}, f_{\texttt{env}})\ \texttt{in}$$
$$\ldots$$
$$\texttt{let}\ f_{\texttt{code}} = f\texttt{.0}\ \texttt{in}$$
$$\texttt{let}\ f_{\texttt{env}} = f\texttt{.2}\ \texttt{in}$$
$$f_{\texttt{code}}\ f_{\texttt{env}}\ 3$$

Environment representation is crucial for the performance of the compiled code

Closure Environments

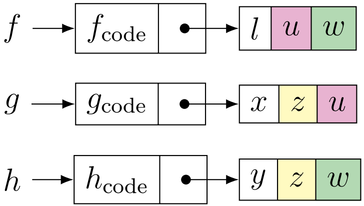
$$
\begin{array}{l}
\texttt{let}\ f\ x\ y\ z = \\
\quad \ldots l \ldots \\
\quad \texttt{let}\ g\ () = \ldots x \ldots z \ldots u \ldots \texttt{in} \\
\quad \texttt{let}\ h\ () = \ldots y \ldots z \ldots w \ldots \texttt{in} \\
\quad h
\end{array}
$$



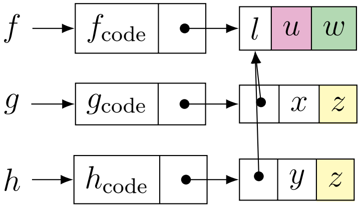
Flat Environments

Closure Environments

$$
\begin{array}{l}
\texttt{let}\ f\ x\ y\ z =\\
\quad \ldots l \ldots\\
\quad \texttt{let}\ g\ () = \ldots x \ldots z \ldots u \ldots \texttt{in}\\
\quad \texttt{let}\ h\ () = \ldots y \ldots z \ldots w \ldots \texttt{in}\\
\quad h
\end{array}
$$



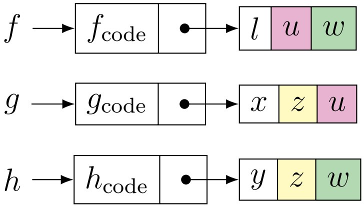
Flat Environments



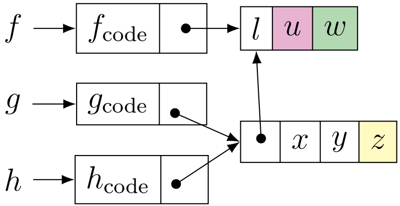
Linked Environments

Closure Environments

$$
\begin{array}{l}
\texttt{let}\ f\ x\ y\ z = \\
\quad \ldots l \ldots \\
\quad \texttt{let}\ g\ () = \ldots x \ldots z \ldots u \ldots \texttt{in} \\
\quad \texttt{let}\ h\ () = \ldots y \ldots z \ldots w \ldots \texttt{in} \\
\quad h
\end{array}
$$



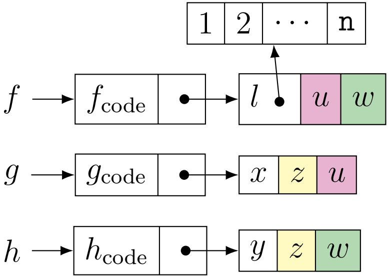
Flat Environments



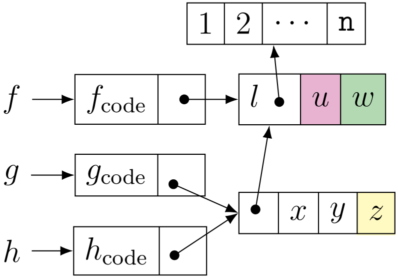
Linked + Shared (JavaScript V8)

Linked and Shared Environments Are Not Safe for Space

$$
\begin{array}{l}
\texttt{let}\ f\ x\ y\ z = \\
\quad \ldots l \ldots \\
\quad \texttt{let}\ g\ () = \ldots x \ldots z \ldots u \ldots \texttt{in} \\
\quad \texttt{let}\ h\ () = \ldots y \ldots z \ldots w \ldots \texttt{in} \\
\quad h
\end{array}
$$



Flat Environments



Linked + Shared (JavaScript V8)

Resource Safety

Theorem

Closure conversion with **flat** *closure environments is safe for* time *and* space

$$cost_{\texttt{trg}} \in \mathcal{O}(cost_{\texttt{src}})$$

Profiling Semantics

$$H; \rho; e \Downarrow_l^{(c,m)} v; H' \qquad \text{for } l \in \{\texttt{src}, \texttt{trg}\}$$

where: $(H, \rho, e) : \texttt{Heap} \times \texttt{Env} \times \texttt{Exp}$ is the input configuration
c is the fuel
m is the space consumption

Fuel based, can throw out-of-time exception: $H; \rho; e \Downarrow_l^{(c_1,m_1)} \bot$

Profiling Semantics

$$H; \rho; e \Downarrow_l^{(c,m)} v; H' \qquad \text{for } l \in \{\texttt{src}, \texttt{trg}\}$$

where: $(H, \rho, e) : \texttt{Heap} \times \texttt{Env} \times \texttt{Exp}$ is the input configuration

c is the fuel

m is the space consumption

Fuel based, can throw out-of-time exception: $H; \rho; e \Downarrow_l^{(c_1,m_1)} \bot$

Source

- Function definition incurs cost proportional to the number of its free variables
- Memory: the maximum size of reachable heap

Target

- Function definition incurs unary cost
- Memory : the maximum size of actual heap
- Invokes an ideal GC upon function entry

Top-level Theorem

Correctness of closure conversion, **closed programs**:

Theorem

If

- $e \leadsto \bar{e}$ *(closure conversion)*
- $e \Downarrow_{\mathtt{src}}^{(c_1,m_1)} v_1; H_1$ *(source evaluation)*

then

- $\bar{e} \Downarrow_{\mathtt{trg}}^{(c_2,m_2)} v_2; H_2$ *(target evaluation)*
- (v_1, H_1) ***relates to*** (v_2, H_2) *(functional correctness)*
- $c_1 \leq c_2 \leq K * c_1$ *(time bound)*
- $m_2 \leq m_1 + \mathtt{cost}^{\mathtt{space}}(e)$ *(space bound)*

Proof

The logical relation (roughly)

$$\{P\}\ (H_1, \rho_1, e_1) \lessapprox^{(k,i)} (H_2, \rho_1, e_2)\ \{Q\}$$

Configuration $(H, \rho, e)\ :\ \texttt{Heap} \times \texttt{Env} \times \texttt{Exp}$

P: precondition on the initial configurations

Q: postcondition on the resource consumption of the two programs

k: step index (bounds execution steps)

i: heap index (bounds heap deapth)

The logical relation (roughly)

$$\{P\}\ (H_1, \rho_1, e_1) \lessapprox^{(k,i)} (H_2, \rho_1, e_2)\ \{Q\}$$

Configuration (H, ρ, e) : $\mathtt{Heap} \times \mathtt{Env} \times \mathtt{Exp}$

P: precondition on the initial configurations

Q: postcondition on the resource consumption of the two programs

Resource consumption preservation
+
Divergence preservation

In conclusion

So far...

- The *first* formal proof that closure conversion is safe for space
- Mechanized in Coq
- General logical relation framework to extend reasoning to intensional properties
- Dead parameter elimination for mut. rec. functions, by Katja Vassilev

In the future...

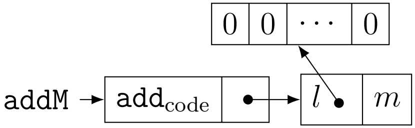
- Extent upwards and downwards
- Propagate through CertiCoq to extend with Coq source logic
- Connect with C translation and concrete GC implementation

Backup Slides

Shared Closures : a space safety counter example

```
fun sum_add (l : list int) : int→int :=
  let sum () = fold (+) l 0 in
  let m = sum () in
  let add n = m + n in
  add

val addM =
 (* [0; ...; 0], M times *)
 let l = repeat 0 M in
 sum_add l
```

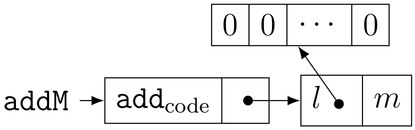


Shared Closures : a space safety counter example

```
fun sum_add (l : list int) : int → int :=
  let sum () = fold (+) l 0 in
  let m = sum () in
  let add n = m + n in
  add

val addM =
 (* [0; ...; 0], M times *)
 let l = repeat 0 M in
 sum_add l
```

Expected space: $\mathcal{O}(1)$
Actual space: $\mathcal{O}(M)$

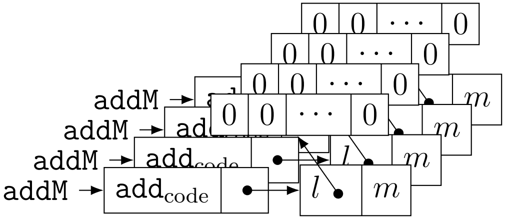


Shared Closures : a space safety counter example

```
fun sum_add (l : list int) : int→int :=
  let sum () = fold (+) l 0 in
  let m = sum () in
  let add n = m + n in
  add

val addM =
 (* [0; ...; 0], M times *)
 let l = repeat 0 M in
 sum_add l
```

Expected space: $\mathcal{O}(M)$
Actual space: $\mathcal{O}(M^2)$



Garbage Collection

$$\mathtt{GC}_S(H_1, H_2, \beta) \overset{\text{def}}{=} \begin{array}{l} S \vdash H_1 \dot{\sim}_\beta H_2 \wedge \\ \mathtt{injective}_{\mathcal{R}(H_1)[S]}(\beta) \wedge \\ \mathtt{dom}(H_2) \subseteq \mathcal{R}(H_2)[\beta(S)] \end{array}$$

H_2 is a collection of H_1 from root set S if:

- The reachable portions of H_1 and H_2 are equivalent up to the injective renaming β
- Only reachable locations are left in the domain ($\mathcal{R}(H)[S]$: set of reachable locations in H from root set S)

CPS IR

$$
\begin{array}{llcrcl}
(\mathit{Expressions}) & e & \in & \texttt{Exp} & ::= & \texttt{let}\ x\ =\ \texttt{C}(\vec{y})\ \texttt{in}\ e \mid \texttt{let}\ x\ =\ y.i\ \texttt{in}\ e \\
 & & & & \mid & \texttt{case}\ y\ \texttt{of}\ \{\texttt{C}_i\ \rightarrow\ e_i\}_{i \in I} \\
 & & & & \mid & \texttt{let}\,\texttt{rec}\ f\ \vec{x}\ =\ e_1\ \texttt{in}\ e_2 \mid f\ \vec{x} \mid \texttt{halt}(x) \\
(\mathit{Locations}) & l & \in & \texttt{Loc} & & \\
(\mathit{Values}) & v & \in & \texttt{Val} & ::= & l \mid \texttt{let}\,\texttt{rec}\ f\ \vec{x}\ =\ e \\
(\mathit{Environments}) & \sigma & \in & \texttt{Env} & = & \texttt{Var} \rightharpoonup \texttt{Loc} \\
(\mathit{Blocks}) & b & \in & \texttt{Block} & ::= & \texttt{C}(\vec{v}) \mid \texttt{Clo}(v, \sigma) \\
(\mathit{Heaps}) & h & \in & \texttt{Heap} & = & \texttt{Loc} \rightharpoonup \texttt{Block}
\end{array}
$$

Semantics: Application rule, pre- closure conversion

$$\frac{\begin{array}{c}\rho(\vec{x}) = \vec{v} \qquad \rho(f) = l \\ H_1(l) = \texttt{Clo}(\texttt{let rec}\ g\ \vec{x}\ =\ e_1, \rho_f) \qquad \texttt{GC}_{\texttt{FL}_{\texttt{Env}}(\rho_f)[\texttt{FV}(e_1)]}(H_1, H_2, \beta) \\ H_2; \beta \circ (\rho_f[\vec{x} \mapsto \vec{v}][g \mapsto l]); e_1 \Downarrow_{\texttt{src}}^{(i-c,m)} r \qquad c \leq i \qquad c = \texttt{cost}(f\ \vec{x})\end{array}}{H_1; \rho; f\ \vec{x} \Downarrow_{\texttt{src}}^{(i,\max(m,\texttt{size}(H_1)))} r}\ \textsc{App}$$

GC happens upon function entry: each function allocates a constant amount of space before the next call

Semantics: Application rule, post- closure conversion

$$\frac{\rho(\vec{x}) = \vec{v} \qquad \rho(f) = \texttt{let}\,\texttt{rec}\ g\ \vec{x}\ =\ e \qquad \texttt{GC}_{\texttt{FL}_{\texttt{Env}}(\rho_f)[\texttt{FV}(e)]}(H_1, H_2, \beta) \\ H_2; \beta \circ ([\vec{x} \mapsto \vec{v}][g \mapsto \texttt{let}\,\texttt{rec}\ g\ \vec{x}\ =\ e]); e \Downarrow_{\texttt{trg}}^{(i-c,m)} r \\ c \leq i \qquad c = \texttt{cost}(f\ \vec{x})}{H_1; \rho; f\ \vec{x} \Downarrow_{\texttt{trg}}^{(i,\max(m,\texttt{size}(H_1)))} r}\ \textsc{App}_{\textsc{cc}}$$