Verification of an ML compiler

Lecture 3:
Closures, closure conversion and call optimisations

Marktoberdorf Summer School MOD 2017

Magnus O. Myreen, Chalmers University of Technology
Implementing the ML abstractions

The two most interesting transitions

function values are implemented (topic of this lecture)

data abstraction is implemented (topic of previous lecture)
Value type before:

\[
\text{v} = \begin{cases}
\text{Number int} \\
\text{Word64 word64} \\
\text{Block num (v list)} \\
\text{RefPtr num} \\
\text{Closure (v list) exp} \\
\text{Recclosure (v list)(exp list) num}
\end{cases}
\]

This is a minor simplification of CakeML’s actual value type here.

Value types after:

\[
\text{v} = \begin{cases}
\text{Number int} \\
\text{Word64 word64} \\
\text{Block num (v list)} \\
\text{RefPtr num} \\
\text{CodePtr num}
\end{cases}
\]

Intermediate languages with first-class functions. No size limits.

No first-class functions. No size limits.
DeBruijn indexing

evaluate ([Var n],env,s) =
  if n < length env then (Rval [el n env],s)
  else (Rerr(Rabort Rtype_error),s)

evaluate ([Let xs x2],env,s) =
  case evaluate (xs,env,s) of
  | (Rval vals,s1) => evaluate ([x2],vals ++ env,s1)
  | res => res)
Semantics of closures

Closure creation in the concrete syntax:

\[ \text{fn } v \Rightarrow e \]

Evaluation in the semantics:

\[ \text{evaluate } ([\text{Fn } e], \text{env}, s) = (\text{Rval } [\text{Closure env } e], s) \]

- No variable name given, since we are using dB indexing
- The created closure captures the current env
Semantics of closures (cont.)

Function application in SML concrete syntax, e.g. fac 50

Evaluation in the semantics:

evaluate ([App e1 e2], env, s) =
  case evaluate env s [e1,e2] of
   | (Rval [f,arg],s1) =>
     (case app_env_exp f arg of
      | Some (env,exp) =>
        if s1.clock = 0 then
        (Rerr (Rabort Rtimeout_error), s1)
        else
        evaluate ([exp], env, dec_clock s1)
      | _ => (Rerr(Rabort Rtype_error),s1))
   | res => res

app_env_exp (Closure env exp) arg = Some ([arg]++env, exp)
This lecture

Function values (called closures) bring challenges:

1. Closures make stating the value relation harder
2. Closures need to be compiled
3. Vital optimisations

*If time allows:* a walk-through of the compiler diagram
This lecture

Function values (called closures) bring challenges:

1. Closures make stating the value relation harder
2. Closures need to be compiled
3. Vital optimisations

*If time allows:* a walk-through of the compiler diagram
Closures cause complications

**Constant folding phase:**

\[
\text{compile } \left( \text{Add } (\text{Lit } i) \ (\text{Lit } j) \right) = \text{Lit } (i+j) \\
\text{compile } \left( \text{Fn } e \right) = \text{Fn } (\text{compile } e) \\
... \\
\]

**Evaluation in the semantics:**

\[
\text{evaluate } \left( \left[ \text{Add } (\text{Lit } 3) \ (\text{Lit } 5) \right], \text{env}, s \right) \\
= (\text{Rval } (\text{Number } 8), s) \\
\text{evaluate } \left( \left[ \text{compile } \left( \text{Add } (\text{Lit } 3) \ (\text{Lit } 5) \right) \right], \text{env}, s \right) \\
= \text{evaluate } \left( \left[ \text{Lit } 8 \right], \text{env}, s \right) \\
= (\text{Rval } (\text{Number } 8), s)
\]

Optimised code produced the same result — **good!**
Closures cause complications

**Constant folding phase:**

\[
\text{compile } (\text{Add } (\text{Lit } i) (\text{Lit } j)) = \text{Lit } (i+j) \\
\text{compile } (\text{Fn } e) = \text{Fn } (\text{compile } e) \\
\]

... 

**Evaluation in the semantics:**

\[
\text{evaluate } ([\text{Fn } (\text{Add } (\text{Lit } 3) (\text{Lit } 5))], \text{env}, s) \\
= ??? \\
\]

\[
\text{evaluate } ([\text{compile } (\text{Fn } (\text{Add } (\text{Lit } 3) (\text{Lit } 5)))]], \text{env}, s) \\
= \text{evaluate } ([\text{Fn } (\text{Lit } 8)], \text{env}, s) \\
= ??? \\
\]
Closures cause complications

**Constant folding phase:**

\[
\begin{align*}
\text{compile } (& \text{Add } (\text{Lit } i) \text{ (Lit } j)) &= \text{Lit } (i+j) \\
\text{compile } &\text{(Fn e)} = \text{Fn } (\text{compile } e) \\
\end{align*}
\]

... 

**Evaluation in the semantics:**

\[
\begin{align*}
\text{evaluate } ([\text{Fn } (\text{Add } (\text{Lit } 3) \text{ (Lit } 5))], \text{env}, \text{s}) &= (\text{Rval } (\text{Closure env } (\text{Add } (\text{Lit } 3) \text{ (Lit } 5))), \text{s}) \\
\text{evaluate } ([\text{compile } (\text{Fn } (\text{Add } (\text{Lit } 3) \text{ (Lit } 5)))], \text{env}, \text{s}) &= \text{evaluate } ([\text{Fn } (\text{Lit } 8)], \text{env}, \text{s}) \\
&= (\text{Rval } (\text{Closure env } (\text{Lit } 8)), \text{s})
\end{align*}
\]

Values can no longer be compared with equality
Value relation options

How do we relate values in presence of closures?

Closure env (Add (Lit 3) (Lit 5))

Closure env (Lit 8)

**Syntactic option:**

\[
\text{val_rel_list } \text{env1 } \text{env2 } \quad \text{e2 } = \text{compile } \text{e1}
\]

\[
\text{val_rel } (\text{Closure env1 } \text{e1}) (\text{Closure env2 } \text{e2})
\]

**Semantic option:**

One can define a *logical relation* which relates closures, if related inputs produce related outputs.

code in closure must be produced by the current compiler function

*jargon*: type-directed, step indexed, …
One can define a logical relation which relates closures, if related inputs produce related outputs.

**Definition:**

\[
\begin{align*}
\text{val\_rel} \ x \ y & \quad \text{val\_rel\_list} \ xs \ ys \\
\hline
\text{val\_rel\_list} \ (x::xs) \ (y::ys) & \\
\end{align*}
\]

\[
\begin{align*}
\text{val\_rel\_list} \ [] \ []
\end{align*}
\]

**Syntactic option:**

\[
\begin{align*}
\text{val\_rel\_list} \ \text{env}1 \ \text{env}2 & \quad e2 = \text{compile} \ e1 \\
\hline
\text{val\_rel} \ (\text{Closure} \ \text{env}1 \ e1) \ (\text{Closure} \ \text{env}2 \ e2)
\end{align*}
\]

**Semantic option:**

_**Jargon:** type-directed, step indexed, …

One can define a logical relation which relates closures, if related inputs produce related outputs.
Pros and Cons

**Syntactic option:**

\[
\text{val}_\text{rel}_\text{list} \quad \text{env1} \quad \text{env2} \quad \text{e2} = \text{compile} \quad \text{e1}
\]

\[
\text{val}_\text{rel} \quad (\text{Closure} \quad \text{env1} \quad \text{e1}) \quad (\text{Closure} \quad \text{env2} \quad \text{e2})
\]

**Pro:** easy to set up  **Con:** compiler specific, boilerplate repeated

**Semantic option:**

One can define a *logical relation* which relates closures, if related inputs produce related outputs.

**Pro:** can be expressive  **Con:** can be very hard to set up
This lecture

Function values (called closures) bring challenges:

1. Closures make stating the value relation harder
2. Closures need to be compiled
3. Vital optimisations

*If time allows:* a walk-through of the compiler diagram
Closure conversion

Value type before:

\[ v = \]
\[ \begin{array}{l}
\text{Number int} \\
\text{Word64 word64} \\
\text{Block num (v list)} \\
\text{RefPtr num} \\
\text{Closure (v list) exp} \\
\text{Recclosure (v list)(exp list) num}
\end{array} \]

Value types after:

\[ v = \]
\[ \begin{array}{l}
\text{Number int} \\
\text{Word64 word64} \\
\text{Block num (v list)} \\
\text{RefPtr num} \\
\text{CodePtr num}
\end{array} \]

Intermediate languages with first-class functions. No size limits.

No first-class functions. No size limits.
Closure conversion

Value type before:

\[
\begin{array}{c}
\text{v} = \\
\text{Number int} \\
\text{Word64 word64} \\
\text{Block num (v list)} \\
\text{RefPtr num} \\
\text{Closure (v list) exp} \\
\text{Recclosure (v list)(exp list) num}
\end{array}
\]

Value types after:

\[
\begin{array}{c}
\text{v} = \\
\text{Number int} \\
\text{Word64 word64} \\
\text{Block num (v list)} \\
\text{RefPtr num} \\
\text{CodePtr num}
\end{array}
\]

Closure values will be represented as tuples with a code pointer.
Value relation

environment list must related to values stored in Block

the compiled code for the body must be in the global code store

\[
\text{val\_rel\_list\ code\ env\ vals} \quad \text{lookup\ p\ code} = \text{compile\ body}
\]

\[
\text{val\_rel\ code\ (Closure\ env\ body)}
\quad (\text{Block\ clos\_tag\ ([\text{CodePtr\ p}]\ ++\ vals)}))
\]

references or internal pointers are used

the Block has a special marker so that equality can distinguish closures from other data

Notes: mutually recursive closures are more complicated to represent because they need to have each other in the env
Minimal environments?

- env and vals are lists of same length

val_rel_list code env vals  lookup p code = compile body

val_rel code (Closure env body)
  (Block clos_tag ([CodePtr p] ++ vals))

Reminder:

evaluate ([Fn e],env,s) = (Rval [Closure env e],s)

Yes! The env can contain values that are never used in e.
Minimal environments?

val_rel_list code env vals   lookup p code = compile body

val_rel code (Closure env body)
(Block clos_tag ([CodePtr p] ++ vals))

Note: any descent compiler will shrink the environments that are stored into the Blocks

CakeML implements this as a compiler phase right before closure conversion
This lecture

Function values (called closures) bring challenges:

1. Closures make stating the value relation harder
2. Closures need to be compiled
3. Vital optimisations

*If time allows:* a walk-through of the compiler diagram
Optimisations with high impact

The optimisations are applied additively from left to right. (Bottom) Comparison of average execution times across ML implementations, relative to OCaml. The error bars show the maximum/minimum times measured over 10 executions.

Of the code before they can apply M-like optimisations, since the standard requires left-to-right evaluation order. Note also that the CakeML compiler uses bignum arithmetic for all of its computations, while most of the other compilers (except Poly/ML) default to fixed sized integers.


these optimisations combined reduce running time by 60% or more
Comparing ML compilers

... and are crucial in making CakeML perform well

execution time relative to native-code compiled OCaml (red)
What do the optimisations do?

*Answer:* improve compilation of closures and calls

in fact, they try to avoid closures if possible
Naive implementations are slow

Example:

```plaintext
fun foo x y z = x+y+z;
val n = foo 0 89 21;
```

The above is syntactic sugar for:

```plaintext
val foo = fn x => (fn y => (fn z => x+y+z));
val n = ((foo 0) 89) 21;
```

we are looking up the value for foo, even though it is possible to known statically

each application only consumes one argument at a time

… between each application a new closure is created
Optimisation of function calls

fun reverse xs = let
  fun append xs ys =
    case xs of [] => ys
    | (x::xs) => x :: append xs ys;
  fun rev xs =
    case xs of [] => xs
    | (x::xs) => append (rev xs) [x]
  in rev xs end;
val example = reverse [1,2,3];
Optimisation of function calls

```ml
set_global 0 (fn xs => let
  fun append xs = fn ys =>
    if xs = [] then ys else
      el 0 xs :: (append (el 1 xs)) ys
  fun rev xs =
    if xs = [] then xs else
      (append (rev (el 1 xs))) [el 0 xs]
  in rev xs end end);
set_global 1 ((get_global 0) [1,2,3]);
```

Optimisation of function calls

set_global 0 (fn \(n_4\) xs => let
  fun append\(0\) \(\langle xs,ys \rangle \) =
    if \(xs = []\) then \(ys\) else
    el \(0\) \(xs\) :: append\(0\) \(\langle el \ 1 \ xs, \ ys \rangle \)
  fun rev\(2\) \(xs\) =
    if \(xs = []\) then \(xs\) else
    append\(0\) \(\langle rev\(2\) (el \ 1 \ xs), \ [el \ 0 \ xs] \rangle \)
    in rev\(2\) \(xs\) end);
set_global 1 ((get_global 0)\(4\) \([1,2,3]\));

- subscripts give each closure body a unique number
- superscripts indicate that a known body is called

true multi-argument closure
Optimisation of function calls

set_global 0 (fn xs => Call 5 \(\langle xs \rangle\));
set_global 1 (Call 5 \([1,2,3]\));

Code Table:
1 \(\langle xs, ys \rangle \Rightarrow \) if \(xs = []\) then \(ys\) else
   el 0 xs ::: Call 1 \(\langle el 1 \, xs, ys \rangle\)

3 \(\langle xs \rangle \Rightarrow \) if \(xs = []\) then \(xs\) else
   Call 1 \(\langle Call 3 \,(el 1 \, xs), \,[el 0 \, xs]\rangle\)

5 \(\langle xs \rangle \Rightarrow \) let
   val append = 0
   val rev = 0
in Call 3 \(\langle xs \rangle\) end

C-like function calls
This lecture

*If time allows:* a walk-through of the compiler diagram
Latest version:

12 intermediate languages (ILs)
and many within-IL optimisations
each IL at the right level of abstraction

for the benefit of proofs and compiler implementation

(next slide zooms in)
Values used by the semantics

Values

Languages

Compiler transformations

source syntax

Parse concrete syntax

source AST

Infer types, exit if fail

no modules

Eliminate modules

no cons names

Replace constructor names with numbers

no declarations

Reduce declarations to exps; introduce global vars

exhaustive pat. matches

Make patterns exhaustive

no pat. match

Move nullary constructor patterns upwards

ClosLang: last language with closures (has multi-arg closures)

Compile pattern matches to nested Ifs and Lets

Rephrase language

Fuse function calls/apps into multi-arg calls/apps

Track where closure values flow; annotate program

Introduce C-style fast calls wherever possible

Remove deadcode

Both proved sound and complete.

Parser and type inferencer as before

Early phases reduce the number of language features

Language with multi-argument closures

Both proved sound and complete.
Compiler transformations

source syntax

source AST

Languages

Values

Parse concrete syntax

abstract values incl. closures

abstract values incl. ref and code pointers

ClosLang: last language with closures (has multi-arg closures)

BVL: functional language without closures

only 1 global, handle in call

DataLang: imperative language

WordLang: imperative language

Language with multi-argument closures

Simple first-order functional language

Imperative language

Machine-like types

Infer types, exit if fail

Eliminate modules

Replace constructor names with numbers

Reduce declarations to exps; introduce global vars

Make patterns exhaustive

Compile pattern matches to nested Ifs and Lets

Rephrase language

Fuse function calls/apps into multi-arg calls/apps

Track where closure values flow; annotate program

Introduce C-style fast calls wherever possible

Remove deadcode

Prepare for closure conv.

Perform closure conv.

Inline small functions

Fold constants and shrink Lets

Split over-sized functions into many small functions

Compile global vars into a dynamically resized array

Optimise Let-expressions

Switch to imperative style

Reduce caller-saved vars

Combine adjacent memory allocations

Remove data abstraction

Simplify program

Select target instructions

Perform SSA-like renaming

Force two-reg code (if req.)

Allocate register names

Concretise stack

Implement GC primitive

Turn stack access into memory accesses

Rename registers to match arch registers/conventions

Flatten code

Delete no-ops (Tick, Skip)

Encode program as concrete machine code

BVL: functional language without closures

only 1 global, handle in call

DataLang: imperative language

WordLang: imperative language

Machine-like types

all languages communicate with the external world via a byte-array-based foreign-function interface.
Compiler transformations

Languages

Values

machine words and code labels

48-bit 32-bit words

No modules
No cons names
No declarations
Exhaustive pattern matches

abstract values incl. closures and ref pointers

abstract values incl. ref and code pointers

64-bit words

32-bit words

ClosLang:
last language with closures
(has multi-arg closures)

Infer types, exit if fail
Eliminate modules
Replace constructor names with numbers
Reduce declarations to exps; introduce global vars
Make patterns exhaustive
Compile pattern matches to nested Ifs and Lets
Rephrase language
Track where closure values flow; annotate program
Fuse function calls/applications into multi-arg calls/applications
Introduce C-style fast calls wherever possible
Remove deadcode
Prepare for closure conversion
Perform closure conversion
Inline small functions
Fold constants and shrink Lets
Split over-sized functions into many small functions
Compile global vars into a dynamically resized array
Optimise Let-expressions
Switch to imperative style
Remove deadcode
Combine adjacent memory allocations
Remove data abstraction
Simplify program
Select target instructions
Perform SSA-like renaming
Force two-reg code (if req.)
Remove deadcode
Allocate register names
Concretise stack
Implement GC primitive
Turn stack access into memory accesses
Rename registers to match arch registers/conventions
Flatten code
Delete no-ops (Tick, Skip)
Encode program as concrete machine code

Imperative compiler with an FP twist: garbage collector, live-var annotations, fast exception mechanisms (for ML)

Targets 5 architectures

All languages communicate with the external world via a byte-array-based foreign-function interface.
What we learnt

Function values (called closures) bring challenges:

1. **Closures make stating the value relation harder** because closure values contain code, which is modified by the compiler.
   
   comparing values with = does not work

2. **Closures need to be compiled to tuples** that carry a code pointer and a snapshot of the relevant environment.

3. **Good compilation** of closures and function applications is crucial for performance.
Extra slides
Mutually recursive closures

Value:

\[ V = \ldots \]

\[
| \text{Recclosure} \left( v \ \text{list} \right) \left( \text{exp} \ \text{list} \right) \text{num} \]

Closure creation in the concrete syntax:

let fun f1 x = ... and f2 = ... and f3 = ... in ... end

Evaluation in the semantics:

\[
\text{evaluate} \left( \left[ \text{Letrec} \ funs \ rest \right], \text{env}, s \right) = \\
\text{evaluate} \left( \left[ \text{rest} \right], \text{build} \ \text{env} \ \text{funs} \ +\ + \ \text{env}, s \right)
\]

\[
\text{build} \ \text{env} \ \text{fns} = \text{Genlist} \left( \text{Recclosure} \ \text{env} \ \text{fns} \right) \left( \text{length} \ \text{fns} \right)
\]

Genlist \ f \ n = \text{if } 0 \ \text{then } [\text{} ] \ \text{else Genlist} \ f \ (n-1) \ +\ + \ [f \ (n-1)]
**Application**

```
evaluate ([App e1 e2], env, s) =
  case evaluate env s [e1, e2] of
  | (Rval [f, arg], s1) =>
    (case app_env_exp f arg of
      | Some (env, exp) =>
        if s1.clock = 0 then
          (Rerr (Rabort Rtimeout_error), s1)
        else
          evaluate ([exp], env, dec_clock s1)
      | _ => (Rerr(Rabort Rtype_error), s1))
  | res => res

app_env_exp (Closure env exp) arg = Some ([arg]++env, exp)

app_env_exp (Recclosure env funs index) arg =
  if index < length funs then
    Some ([arg] ++ build env funs ++ env, el index funs)
  else None
```