Proving PureCake (and CakeML)

Erik Oosting

Papers

- **The Verified CakeML Compiler Backend**
  - (https://doi.org/10.1017/S0956796818000229)
- **PureCake: A verified compiler for a lazy functional language**
  - (https://doi.org/10.1145/3591259)

Outline

- **Introduction**
- **CakeML (Compilation)**
- **PureCake Evaluation**
- **PureCake Compilation**

About CakeML

- **Formally verified compiler for a dialect of Standard ML**
- **Takes resource constraints into account for it's proof of correct compilation**
- **Implemented in HOL4**
    - Theorems and proofs as data
    - Simply typed

```
fun fac n = if n = 0 then 1 else fac (n-1) * n;

fun main () =
  let
    val arg = List.hd (CommandLine.arguments())
    val n = Option.valOf (Int.fromString arg)
  in
    print_int (fac n) ; print "\n"
  end
  handle _ =>
    TextIO.print_err ("usage: " ^ CommandLine.name() ^ " <n>\n");

main ();
```

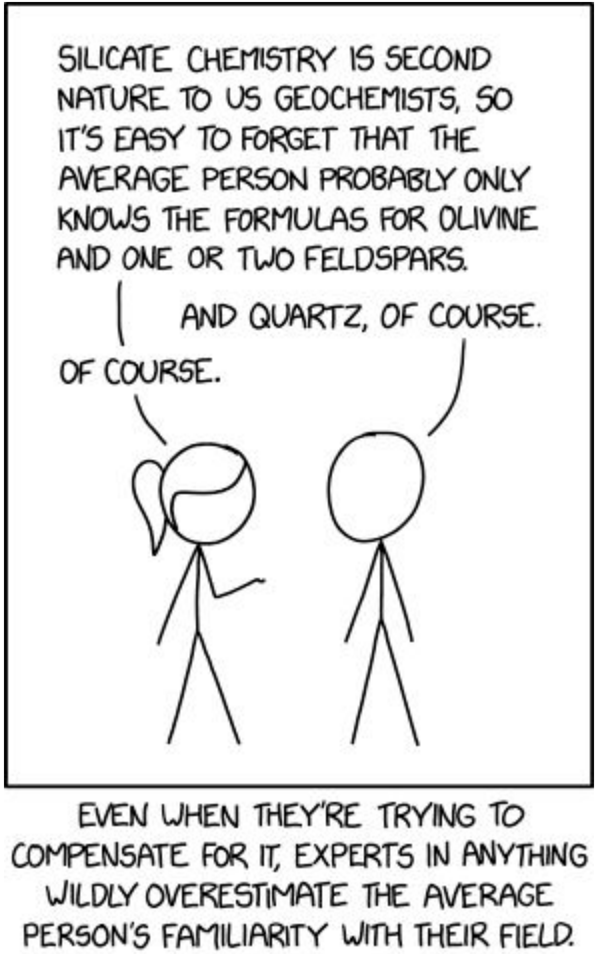
About PureCake

- **Lazily evaluated**
- **A bit more complicated**

```
numbers :: [Integer]
numbers =
  let num n = n : num (n + 1)
  in num 0

factA :: Integer -> Integer -> Integer
factA a n =
  if n < 2 then a
  else factA (a * n) (n - 1)

factorials :: [Integer]
factorials = map (factA 1) numbers

app :: (a -> IO b) -> [a] -> IO ()
app f l = case l of
            []  -> return ()
            h:t -> do f h ; app f t

main :: IO ()
main = do
  arg1 <- read_arg1
  -- fromString == 0 on malformed input
  let i = fromString arg1
      facts = take i factorials
  app (\i -> print $ toString i) facts
```

About this presentation

- **These papers are about a lot of *stuff***
  - CakeML and PureCake are big projects
  - A lot of compiler techniques are used to get to where we are
- **I like reading about compilers a lot...**



How to make a compiler

- **Figure out what your starting (source) language does**
  - What makes a program in the starting language correct?
  - What outside behaviours does it have?
- **Figure out what your target language does**
  - What does it do different from the source language?
  - How are you going to wrangle the behaviour of the source language into the target language?
- **Recommended: make more compilers**
  - Define *Intermediate Languages*
  - Put the compilers for all your intermediate languages together

How to prove a compiler

- **Make sure all programs “behave as expected”**
  - No memory leaks
  - “Semantics” between source and target stay the same

Context: CakeML

CakeML Design Goals

- **Approachable to newcomers**
  - Easily extensible
  - Usable for future research/student projects
- **Keep the computer in mind**
  - Computers don't have infinite memory. We can run out!
- ***Juusst* the right number of intermediate languages**
  - Too many and we have a lot of unnecessary work
  - Too little and the compilation steps become too convoluted to prove

Context: CakeML

I/O effects

$$\mathsf{semantics} : \varphi\ \texttt{ffi\_state} \rightarrow \texttt{program} \rightarrow \texttt{behaviour}\ \texttt{set}$$

$$\begin{aligned}\texttt{behaviour} &= \mathsf{Diverge}\ (\texttt{io\_event}\ \texttt{stream}) \mid \mathsf{Terminate}\ \texttt{outcome}\ (\texttt{io\_event}\ \texttt{list}) \mid \mathsf{Fail} \\ \texttt{outcome} &= \mathsf{Success} \mid \mathsf{Resource\_limit\_hit} \mid \mathsf{FFI\_outcome}\ \texttt{final\_event}\end{aligned}$$

The CakeML compilation pipeline

General Compiler proofs

We need a correctness proof for every compilation step.

- ***config*** **-> Arbitrary machine config**
- **"syntactic_condition" -> no errors in the program**

$$\vdash \textsf{compile}\ \mathit{config}\ \mathit{prog} = \mathit{new\_prog}\ \wedge$$
$$\textsf{syntactic\_condition}\ \mathit{prog}\ \wedge$$
$$\textsf{Fail} \notin \textsf{semantics}_{\mathrm{A}}\ \mathit{ffi}\ \mathit{prog} \Rightarrow$$
$$\textsf{semantics}_{\mathrm{B}}\ \mathit{ffi}\ \mathit{new\_prog} = \textsf{semantics}_{\mathrm{A}}\ \mathit{ffi}\ \mathit{prog}$$

General Compiler proofs

The program may run out of memory:

$$\text{semantics}_{\text{B}}\ \textit{ffi}\ \textit{new\_prog} \subseteq \text{extend\_with\_resource\_limit}\ (\text{semantics}_{\text{A}}\ \textit{ffi}\ \textit{prog})$$

$$\begin{array}{l}\text{extend\_with\_resource\_limit}\ \textit{behaviours} = \\ \quad \textit{behaviours} \cup \\ \quad \{\ \text{Terminate Resource\_limit\_hit}\ \textit{io\_list} \mid \exists t\ l.\ \text{Terminate}\ t\ l \in \textit{behaviours} \wedge \textit{io\_list} \preccurlyeq l\ \} \cup \\ \quad \{\ \text{Terminate Resource\_limit\_hit}\ \textit{io\_list} \mid \exists ll.\ \text{Diverge}\ ll \in \textit{behaviours} \wedge \text{fromList}\ \textit{io\_list} \preccurlyeq_{\infty} ll\ \}\end{array}$$

Parsing to an AST

- **Using a Parsing Expression Grammar**
    - Order sensitive
    - Non-terminals have a *rank* based on the input they consume
    - Ranks ensure that the parser consume input
- **Type inference**
    - Uses “triangular substitution”
    - No let-polymorphism
- **Removing syntactic language features**
    - No modules, ADTs, incomplete pattern matches, *names*
- **Result: A fully typed, nameless, simple programming language**

```
exp =
   Var num
 | If exp exp exp
 | Let (exp list) exp
 | Raise exp
 | Handle exp exp
 | Tick exp
 | Call num (num option) (exp list)
 | Op op (exp list)
```

Compilation pipeline

CLOSLang

```
v =
    Number int
  | Word64 (64 word)
  | Block num (v list)
  | ByteVector (8 word list)
  | RefPtr num
  | Closure (num option) (v list) (v list) num exp
  | Recclosure (num option) (v list) (v list) ((num × exp) list) num
```

- **Functions -> Closures**
- **Used for lambda lifting**
- **Closures:**
  - (Optional) location of the closure
  - Evaluation environment (values for free variables ‘Var’ in the environment)
  - Arguments already passed to the closure
  - Number of arguments the closure still needs
  - The closure body
- **Recursive closures**
  - Same as closures, except this time a list of needed arguments and function bodies
  - Finally, a list index indicating where to start evaluation

```
exp =
    Var num
  | If exp exp exp
  | Let (exp list) exp
  | Raise exp
  | Handle exp exp
  | Tick exp
  | Call num (num option) (exp list)
  | Op op (exp list)
```

The ByteVectorLangauge (BVL)

- **No closures!**
- **Type checking happens here**
- **'Closure' ->** Block closure_tag ([CodePtr *ptr*; Number *arg_count*] ++ *free_var_vals*)
- **'RecClosure' ->** Block closure_tag [CodePtr *ptr*; Number *arg_count*; RefPtr *ref_ptr*]

```
v =
    Number int
  | Word64 (64 word)
  | Block num (v list)
  | CodePtr num
  | RefPtr num

exp =
    Var num
  | If exp exp exp
  | Let (exp list) exp
  | Raise exp
  | Handle exp exp
  | Tick exp
  | Call num (num option) (exp list)
  | Op op (exp list)
```

```
evaluate ([],env,s) = (Rval [],s)

evaluate (x::y::xs,env,s) =
  case evaluate ([x],env,s) of
    (Rval v1,s1) ⇒
      (case evaluate (y::xs,env,s1) of
         (Rval vs,s2) ⇒ (Rval (v1 ++ vs),s2)
       | (Rerr e,s2) ⇒ (Rerr e,s2))
  | (Rerr v10,s1) ⇒ (Rerr v10,s1)

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

evaluate ([Let xs x],env,s) =
  case evaluate (xs,env,s) of
    (Rval vs,s1) ⇒ evaluate ([x],vs ++ env,s1)
  | (Rerr e,s1) ⇒ (Rerr e,s1)

evaluate ([Op op xs],env,s) =
  case evaluate (xs,env,s) of
    (Rval vs,s1) ⇒
      (case do_app op (rev vs) s1 of
         Rval (v,s2) ⇒ (Rval [v],s2)
       | Rerr err ⇒ (Rerr err,s1))
  | (Rerr v9,s1) ⇒ (Rerr v9,s1)

evaluate ([Raise x],env,s) =
  case evaluate ([x],env,s) of
    (Rval vs,s1) ⇒ (Rerr (Rraise (hd vs)),s1)
  | (Rerr e,s1) ⇒ (Rerr e,s1)
```

```
evaluate ([Handle x1 x2],env,s) =
  case evaluate ([x1],env,s) of
    (Rval v,s1) ⇒ (Rval v,s1)
  | (Rerr (Rraise v),s1) ⇒ evaluate ([x2],v::env,s1)
  | (Rerr (Rabort e),s1) ⇒ (Rerr (Rabort e),s1)

evaluate ([Call ticks dest xs],env,s) =
  case evaluate (xs,env,s) of
    (Rval vs,s1) ⇒
     (case find_code dest vs s1.code of
        None ⇒ (Rerr (Rabort Rtype_error),s1)
      | Some (args,exp') ⇒
          if s1.clock < ticks + 1 then
            (Rerr (Rabort Rtimeout_error),s1 with clock := 0)
          else
            evaluate ([exp'],args,dec_clock (ticks + 1) s1))
  | (Rerr v8,s1) ⇒ (Rerr v8,s1)
   ...

do_app (Const i) [] s = Rval (Number i,s)

do_app (Cons tag) xs s = Rval (Block tag xs,s)
   ...
```

DATAlang

- **Turning BVL into an imperative language**
- **Semi-manual GC with ‘MakeSpace’**
- **‘num’s are variables**
- **Used for optimizations in memory allocations**

```
prog =
    Skip
  | Move num num
  | Call ((num × num_set) option) (num option)
        (num list) ((num × prog) option)
  | Assign num op (num list) (num_set option)
  | Seq prog prog
  | If num prog prog
  | MakeSpace num num_set
  | Raise num
  | Return num
  | Tick
```

DATALang

- **Explicit call stack**
- **More direct error handling**

$$
\begin{aligned}
&\varphi\ \texttt{state} = \langle \\
&\quad \mathsf{locals} : \texttt{v num\_map}; \\
&\quad \mathsf{stack} : \texttt{frame list}; \\
&\quad \mathsf{global} : \texttt{num option}; \\
&\quad \mathsf{handler} : \texttt{num}; \\
&\quad \mathsf{refs} : \texttt{num} \mapsto \texttt{v ref}; \\
&\quad \mathsf{clock} : \texttt{num}; \\
&\quad \mathsf{code} : (\texttt{num} \times \texttt{prog})\ \texttt{num\_map}; \\
&\quad \mathsf{ffi} : \varphi\ \texttt{ffi\_state}; \\
&\quad \mathsf{space} : \texttt{num} \\
&\rangle
\end{aligned}
$$

$$
\begin{aligned}
\texttt{v} =\ & \mathsf{Number}\ \texttt{int} \\
\mid\ & \mathsf{Word64}\ (64\ \texttt{word}) \\
\mid\ & \mathsf{Block}\ \texttt{num}\ (\texttt{v list}) \\
\mid\ & \mathsf{CodePtr}\ \texttt{num} \\
\mid\ & \mathsf{RefPtr}\ \texttt{num}
\end{aligned}
$$

$$\texttt{frame} = \mathsf{Env}\ (\texttt{v num\_map}) \mid \mathsf{Exc}\ (\texttt{v num\_map})\ \texttt{num}$$

$$\alpha\ \texttt{ref} = \mathsf{ValueArray}\ (\alpha\ \texttt{list}) \mid \mathsf{ByteArray}\ \texttt{bool}\ (8\ \texttt{word list})$$

Intermezzo: CakeML Evaluation

Explaining Evaluation using ANF

- (Sabry & Feleissen, 1992)
- **Implicit in the BVL step**
- **ANF separates nested function calls into ‘let’ bindings TODO FIX THIS**

Original

```
EXP ::= λ VAR . EXP
      | EXP EXP
      | VAR
      | CONST
      | let VAR = EXP in EXP

CONST ::= f | g | h
```

ANF

```
EXP ::= VAL
      | let VAR = VAL in EXP
      | let VAR = VAL VAL in EXP

VAL ::= VAR
      | CONST
      | λ VAR . EXP

CONST ::= f | g | h
```

Explaining Evaluation using ANF

- **Some more practical grammars**

```
%% Normal expressions
EXP ::= λ var . EXP
      | EXP(EXP, ...)
      | VAR
      | CONST
      | EXP + EXP | EXP - EXP
      | EXP * EXP | EXP / EXP
      | let VAR = EXP in EXP
      | if EXP then EXP else EXP

CONST ::= 0 | 1 | 2 ...
```

```
%% ANF grammar
EXP ::= VAL
      | let VAR = VAL in EXP
      | let VAR = VAL + VAL in EXP
      | let VAR = VAL - VAL in EXP
      | let VAR = VAL * VAL in EXP
      | let VAR = VAL / VAL in EXP
      | let VAR = VAL(VAL, ...)
      | if VAL then EXP else EXP

VAL ::= λ VAR . EXP
      | CONST
      | VAR
```

Intermezzo

Explaining Evaluation using ANF

```
def fac n = if n == 0 then 1 else fac(n - 1) * n
```

```
def fac n =
    let b = n == 0 in
    if b then 1 else (let n'  = n - 1     in
                      let acc = fac (n') in
                      n * acc)
```

PureCake

About PureCake

- **Looks like Haskell**
- ***Works**** **like Haskell**
  - Has lazy evaluation
  - And substitution semantics
- **Formalizes some of the things CakeML is using**
- **Compiles to CakeML**

About Purecake

$$\mathsf{exp\_of}\,(\mathbf{case}\;x = ce\;\mathbf{of}\;\overline{row_n}) \stackrel{\text{def}}{=} \mathbf{let}\;x = \mathsf{exp\_of}\;ce\;\mathbf{in}\;\mathsf{expand}_x\;[\,\overline{row_n}\,]$$

$$\mathsf{expand}_x\;[\mathit{cname}[\,\overline{y_n}\,] \rightarrow ce',\;\overline{row_m}\,] \stackrel{\text{def}}{=} \mathbf{if}\;(\mathbf{eq}_?\;\mathit{cname}\;n\;(\mathbf{var}\;x))\;\mathbf{then}$$

$$\mathbf{let}\;\overline{y_n = \mathbf{proj}_n\;\mathit{cname}\;(\mathbf{var}\;x)}\;\mathbf{in}\;(\mathsf{exp\_of}\;ce')$$

$$\mathsf{expand}_x\;[\,] \stackrel{\text{def}}{=} \mathbf{fail} \qquad \mathbf{else}\;\mathsf{expand}_x\;[\,\overline{row_m}\,]$$

$$
\begin{aligned}
op ::=\; & \mid \mathbf{cons}\;\mathit{cname} \\
& \mid \mathbf{tuple} \\
& \mid \mathbf{prim}\;\mathit{primop} \\
& \mid \mathbf{monadic}\;\mathit{mop}
\end{aligned}
$$

$$
\begin{aligned}
ce ::=\; & \mid \mathbf{var}\;x \\
& \mid op[\,\overline{ce_n}\,] \\
& \mid \lambda\,\overline{x_n}\,.\;ce \\
& \mid ce \cdot \overline{ce_n} \\
& \mid \mathbf{let}\;x = ce_1\;\mathbf{in}\;ce_2 \\
& \mid \mathbf{letrec}\;\overline{x_n = ce_n}\;\mathbf{in}\;ce \\
& \mid \mathbf{seq}\;ce_1\;ce_2 \\
& \mid \mathbf{case}\;x = ce\;\mathbf{of}\;\overline{\mathit{cname}_n[\,\overline{x_{n\,m}}\,] \rightarrow ce_n}
\end{aligned}
$$

$$
\begin{aligned}
e ::=\; & \mid \mathbf{var}\;x \\
& \mid op[\,\overline{e_n}\,] \\
& \mid \lambda x.\;e \\
& \mid e_1 \cdot e_2 \\
& \mid \mathbf{let}\;x = e_1\;\mathbf{in}\;e_2 \\
& \mid \mathbf{letrec}\;\overline{x_n = e_n}\;\mathbf{in}\;e \\
& \mid \mathbf{seq}\;e_1\;e_2 \\
& \mid \mathbf{if}\;e\;\mathbf{then}\;e_1\;\mathbf{else}\;e_2 \\
& \mid \mathbf{eq}_?\;\mathit{cname}\;\mathit{arity}\;e \\
& \mid \mathbf{proj}_n\;\mathit{cname}\;e
\end{aligned}
$$

PureCake: Evaluation

I/O Effects (Interaction trees)

- **Unlike CakeML, we want to model all possible interactions with the outside world**
  - We can use Interaction Trees (Li-yao Xia et al. 2019)
- **Co-inductive datatype that can represent all kinds of semantics**

$$\text{itree } E\ R ::= \text{Ret } (r : R) \mid \text{Tau } (t : \text{itree } E\ R) \mid \text{Vis } (A : \text{Type})\ (e : E\ A)\ (k : A \longrightarrow \text{itree } E\ R)$$

$$\text{itree } E\ A\ R ::= \text{Ret } (r : R) \mid \text{Div} \mid \text{Vis } (e : E)\ (k : A \longrightarrow \text{itree } E\ A\ R)$$

Continuations

All programs have a past, a present and a future

- **The past:**
  - Variables
  - Assigned memory
- **The present**
  - The expression we're currently evaluating
  - The instruction we're currently running
- **The future**
  - Return pointers etc.
  - Continuations!
- **Continuations are modeled as a function, with the current expression result as input, and the program result as output**

I/O Effects (Interaction trees)

$$\mathsf{itree}\ E\ A\ R ::= \mathsf{Ret}\ (r : R) \mid \mathsf{Div} \mid \mathsf{Vis}\ (e : E)\ (k : A \rightarrow \mathsf{itree}\ E\ A\ R)$$

$$\begin{aligned} wh ::= & \\ \mid\ & \textbf{constructor}\ \mathit{cname}[\ \overline{e_n}\ ] \\ \mid\ & \textbf{tuple}\ [\ \overline{e_n}\ ] \\ \mid\ & \textbf{monadic}\ \mathit{mop}[\ \overline{e_n}\ ] \\ \mid\ & \textbf{lambda}\ x\ e \\ \mid\ & \textbf{literal}\ \mathit{lit} \\ \mid\ & \textbf{error} \\ \mid\ & \textbf{diverge} \end{aligned}$$

$$\begin{aligned} E ::= & \\ \mid\ & \textbf{ffi}\ (ch, s) \\ A ::= & \\ \mid\ & \textbf{ok}\ s \\ \mid\ & \textbf{fail}_{\text{ffi}} \\ \mid\ & \textbf{diverge}_{\text{ffi}} \end{aligned}$$

$$\begin{aligned} R ::= & \\ \mid\ & \textbf{termination} \\ \mid\ & \textbf{error} \\ \mid\ & \textbf{fail}_{\text{ffi}} \\ \mid\ & \textbf{diverge}_{\text{ffi}} \end{aligned}$$

I/O Effects (Interaction trees)

$$\llparenthesis\, \textbf{diverge},\ \kappa,\ \sigma \,\rrparenthesis = \mathsf{Div} \qquad\qquad \llparenthesis\, \textbf{error},\ \kappa,\ \sigma \,\rrparenthesis = \mathsf{Ret}\ \textbf{error}$$

$$\llparenthesis\, \textbf{bind}\ e_1\ e_2,\ \kappa,\ \sigma \,\rrparenthesis = \llparenthesis\, \mathsf{eval}_{\mathrm{wh}}\ e_1,\ \textbf{bind}\ \bullet\ e_2 :: \kappa,\ \sigma \,\rrparenthesis$$

$$\llparenthesis\, \textbf{return}\ e,\ \varepsilon,\ \sigma \,\rrparenthesis = \mathsf{Ret}\ \textbf{termination}$$

$$\llparenthesis\, \textbf{return}\ e_1,\ \textbf{bind}\ \bullet\ e_2 :: \kappa,\ \sigma \,\rrparenthesis = \llparenthesis\, \mathsf{eval}_{\mathrm{wh}}\ (e_2 \cdot e_1),\ \kappa,\ \sigma \,\rrparenthesis$$

$$\llparenthesis\, \textbf{raise}\ e_1,\ \mathit{frame} :: \ldots :: \textbf{handle}\ \bullet\ e_2 :: \kappa,\ \sigma \,\rrparenthesis = \llparenthesis\, \mathsf{eval}_{\mathrm{wh}}\ (e_2 \cdot e_1),\ \kappa,\ \sigma \,\rrparenthesis$$

$$\mathsf{eval}_{\mathrm{wh}}\ e = \textbf{literal}\ (\textbf{loc}\ l) \Rightarrow \llparenthesis\, \textbf{len}\ e,\ \kappa,\ \sigma \,\rrparenthesis = \llparenthesis\, \textbf{return}\ (\textbf{int}\ |\sigma(l)|),\ \kappa,\ \sigma \,\rrparenthesis$$

$$\llparenthesis\, \textbf{action}\ (\textbf{msg}\ \mathit{ch}\ s),\ \kappa,\ \sigma \,\rrparenthesis = \mathsf{Vis}\ (\mathit{ch}, s)\ (\lambda a.\ \ldots)$$

where $\textbf{bind}\ e_1\ e_2 \stackrel{\text{def}}{=} \textbf{monadic bind}[\,e_1\ e_2\,]$, *similarly for other monadic operations above.*

Demand analysis

- **By default, all variables are stored in heap memory**
- **Goal: Make as much as possible eager without affecting semantics**
- **Special case: ‘seq’**
- **Demand analysis says nothing about evaluation order**

```
main :: IO ()
main = do
  arg1 <- read_arg1
  let n = fromString arg1
  print $ "Finding longest Collatz sequence less than " ++ toString n
  let res = maxCollatzSequence n
  print $ "Number with longest sequence: " ++ toString (fst res)
  print $ "Length of sequence: " ++ toString (snd res)
  Ret ()

maxCollatzSequence :: Integer -> (Integer, Integer)
maxCollatzSequence n = maxIndex (take n collatzSequences)

collatzSequences :: [Integer]
collatzSequences = map collatzSequence (numbers 0)

collatzSequence :: Integer -> Integer
collatzSequence n =
  let seqAux acc n =
        if n < 1 then (0-1)
        else if n == 1 then acc
        else seqAux (acc + 1) (collatz n)
  in seqAux 0 n

collatz :: Integer -> Integer
collatz n = if n `mod` 2 == 0 then n `div` 2 else 3 * n + 1

```

Demand analysis

$$\frac{}{C \vdash (\mathbf{var}\ x)\ \text{demands}\ x}$$

$$\frac{C \vdash e_2\ \text{demands}\ x \qquad x \neq y}{C \vdash (\mathbf{let}\ y = e_1\ \ \mathbf{in}\ \ e_2)\ \text{demands}\ x}$$

$$\frac{\begin{array}{c} C \vdash e_1\ \text{demands}\ x \\ C \vdash e_2\ \text{demands}\ y \end{array}}{C \vdash (\mathbf{let}\ y = e_1\ \ \mathbf{in}\ \ e_2)\ \text{demands}\ x}$$

$$\frac{C \vdash e_1\ \text{demands}\ x}{C \vdash (\mathbf{seq}\ e_1\ e_2)\ \text{demands}\ x}$$

$$\frac{C \vdash e_2\ \text{demands}\ x}{C \vdash (\mathbf{seq}\ e_1\ e_2)\ \text{demands}\ x}$$

$$[\![\ \mathbf{let}\ x = \bot\ \ \mathbf{in}\ \ \mathbf{seq}\ \mathbf{fail}\ (\mathbf{var}\ x)\ ]\!] = \text{Ret}\ \mathbf{error}$$

$$[\![\ \mathbf{let}\ x = \bot\ \ \mathbf{in}\ \ \mathbf{seq}\ (\mathbf{var}\ x)\ (\mathbf{seq}\ \mathbf{fail}\ (\mathbf{var}\ x))\ ]\!] = \text{Div}$$

Demand Analysis

- **Things get tricky when analysing functions & function calls though**
- **Three cases:**
  - Applied expressions need to be demanded

$$e \text{ demands}_{f} (n, m) \stackrel{\text{def}}{=} \forall x\ \overline{e_m}.\ e_n \text{ demands } x \Rightarrow (e \cdot \overline{e_m}) \text{ demands } x$$

  - Function arguments need to be demanded when they are applied

$$e \text{ demands}_{\text{wa}} (x, n) \stackrel{\text{def}}{=} \forall \overline{e_n}.\ (e \cdot \overline{e_n}) \text{ demands } x$$

  - The recursive case

$$\left(\forall f'\ ds\ xs\ e'\ d.\ (f', ds, \lambda xs.\ e') \in binds \land d \in ds \Rightarrow (\text{reformulate } binds\ e') \text{ demands } d\right)$$
$$\Rightarrow \textbf{letrec } \{ f = e_f \mid (f, ds, e_f) \in binds \}\ e \approx$$
$$\textbf{letrec } \{ f = \text{mark\_demanded } ds\ e_f \mid (f, ds, e_f) \in binds \}\ e$$

Compiling PureCake

Parsing

- **We have indents now, so no normal CFG**
- **Instead, we add an indentation indicators to our CFGs**

$$|\text{Decl}| \longrightarrow |\text{Ident}|^{=}\ \text{'::'}^{>}\ \text{Ty}^{>}$$

- **We can now calculate the “Indentation sets of non-terminals**
  - Either a closed set of possible indentations (*i* to *j* no. of indents)
  - A lower-bounded set (*i* or more no. of indents)
  - Any number of indents
  - Nowhere (this would be a parsing error)
- **Result is the program AST, represented as a giant letrec-statement**

Type inference

- **Classical Hindley-Milner algorithms give bad error messages**
- **We'll use a constraint-based system instead**

$$\frac{\Gamma \vdash ce_1 : \tau_1 \qquad \overline{\alpha_n} \notin \Gamma \qquad \Gamma, x : \forall \overline{\alpha_n} \,.\, \tau_1 \vdash ce_2 : \tau_2}{\Gamma \vdash \textbf{let}\ x = ce_1\ \textbf{in}\ ce_2 : \tau_2}\ \textsc{HMLet}$$

$$\frac{}{M \vdash \textbf{var}\ x : \alpha \ \Rightarrow\ [x : \alpha]\ ;\ \varnothing}\ \textsc{TopVar}$$

$$\frac{\overline{\alpha_n}, M \vdash ce : \tau' \ \Rightarrow\ A\ ;\ C}{M \vdash (\lambda\, \overline{x_n} \,.\, ce) : (\overline{\alpha_n} \rightarrow \tau') \ \Rightarrow\ A \setminus \overline{x_n}\ ;\ C \cup \bigcup_n \{\tau \equiv \alpha_n \mid x_n : \tau \in A\}}\ \textsc{TopLam}$$

$$\frac{M \vdash ce_1 : \tau_1 \ \Rightarrow\ A_1\ ;\ C_1 \qquad M \vdash ce_2 : \tau_2 \ \Rightarrow\ A_2\ ;\ C_2}{M \vdash (\textbf{let}\ x = ce_1\ \textbf{in}\ ce_2) : \tau_2 \ \Rightarrow\ A_1 \cup A_2 \setminus x\ ;\ C_1 \cup C_2 \cup \{\tau \preceq_M \tau_1 \mid x : \tau \in A_2\}}\ \textsc{TopLet}$$

Type Inference

- **We now have constraints**
- **Constraint solving is “straight-forward” and “omitted”**

Demand Analysis

- **We’ve already done this**
- **Result is adding ‘seq’ to expressions we know we can demand without affecting semantics**

Backend: The ILs

- **Instead of proving semantics preservation with functions, we use relations between different ILs**
    - More flexible than functions
    - Means we don’t need to keep track of compiler invariants between all our functions
- **We can then reconstruct a function out of the relations we’ve made**

Purecake compilation

THUNKLang

- **Very similar to the source language**
- **Eagerly evaluated**
- **2 new constructs**
    - delay: turns an expression into a thunk
    - force: evaluates e to a thunk, then forces evaluation of the thunk
- **Note: at this point a thunk re-evaluates every time it is forced!**

```
delay :: a -> (() -> a)
delay e = \() -> e

force :: (() -> a) -> a
force e = e ()
```

THUNKLang

$$\frac{}{\textbf{var}\ x \xrightarrow{\text{thunk}} \textbf{force}\ (\textbf{var}\ x)}\ \textsc{ThkVar}$$

$$\frac{e_1 \xrightarrow{\text{thunk}} e_1' \qquad e_2 \xrightarrow{\text{thunk}} e_2'}{e_1 \cdot e_2 \xrightarrow{\text{thunk}} e_1' \cdot \textbf{delay}\ e_2'}\ \textsc{ThkApp}$$

$$\frac{e_1 \xrightarrow{\text{thunk}} e_1' \qquad e_2 \xrightarrow{\text{thunk}} e_2'}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \xrightarrow{\text{thunk}} \textbf{let}\ x = \textbf{delay}\ e_1'\ \textbf{in}\ e_2'}\ \textsc{ThkLet}$$

$$\frac{e_1 \xrightarrow{\text{thunk}} e_1' \qquad e_2 \xrightarrow{\text{thunk}} e_2' \qquad \mathit{fresh} \notin \text{freevars}\ e_2}{\textbf{seq}\ e_1\ e_2 \xrightarrow{\text{thunk}} \textbf{let}\ \mathit{fresh} = e_1'\ \textbf{in}\ e_2'}\ \textsc{ThkSeq}$$

THUNKLang

- **Note the simplicity in compilation thanks to demand analysis**
  - However, this translation introduces a lot of ‘delay(force(e))’ constructs
  - Define a relation *unthunk* and prove ‘mk_delay’ satisfies this relation
-

$$\text{mk\_delay}\ ce \stackrel{\text{def}}{=} \begin{cases} \mathbf{var}\ x & \text{if } ce = \mathbf{force}\ (\mathbf{var}\ x), \\ \mathbf{delay}\ ce & \text{otherwise.} \end{cases}$$

EnvLang

- **We use environments, instead of substituting functions with their definitions**

StateLang

- **Compile ‘delay’ and ‘force’ primitives into actual expressions**
  - ‘delay’ computations are stored in a mutable array
  - ‘force’ primitives are possible updates to the mutable array (if the value inside of it hasn’t been forced yet)
- **Monadic operations are also compiled into thunk-style functions**
  - “Stateful operations” (Exceptions, mutable array handling, I/O etc.) are turned into special primitives
  - Other operations are turned into computations that accept a unit input to perform the actual operation.

StateLang

$$\lfloor \textbf{return}\ ce \rfloor \stackrel{\text{def}}{=} \textbf{let}\ x = \lfloor ce \rfloor\ \textbf{in}\ \lambda\_.\ \textbf{var}\ x$$

$$\lfloor \textbf{raise}\ ce \rfloor \stackrel{\text{def}}{=} \textbf{let}\ x = \lfloor ce \rfloor\ \textbf{in}\ \lambda\_.\ \textbf{raise}_{\text{prim}}(\textbf{var}\ x)$$

$$\lfloor \textbf{bind}\ ce_1\ ce_2 \rfloor \stackrel{\text{def}}{=} \lambda\_.\ \lfloor ce_2 \rfloor \cdot (\lfloor ce_1 \rfloor \cdot \textbf{unit}) \cdot \textbf{unit}$$

$$\lfloor \textbf{delay}\ ce \rfloor \stackrel{\text{def}}{=} \textbf{alloc}\ [\textbf{false}, \lambda\_.\ \lfloor ce \rfloor]$$

$$\begin{aligned}\lfloor \textbf{force}\ ce \rfloor \stackrel{\text{def}}{=}\ & \textbf{let}\ x = \lfloor ce \rfloor\ ;\ x_0 = x[0]\ ;\ x_1 = x[1]\ \textbf{in} \\ & \quad \textbf{if}\ \textbf{var}\ x_0\ \textbf{then}\ \textbf{var}\ x_1\ \textbf{else} \\ & \qquad \textbf{let}\ w = (\textbf{var}\ x_1) \cdot \textbf{unit}\ \textbf{in} \\ & \qquad\quad x[0] := \textbf{true}\ ;\ x[1] := \textbf{var}\ w\ ;\ \textbf{var}\ w\end{aligned}$$

StateLang

- **We mostly need to prove that our operations on thunks are correct**
  - We need to prove that semantics preserve ‘EnvLang -> StateLang’ AND ‘StateLang -> Envlang’
- **A bit of cleanup:**
  - Remove cases of ‘(λ().ce)()’ and replace them with ‘ce’
- **Semantics themselves are implemented by a CESK machine**
  - Relatively straight-forward
  - Stateful primitives are implemented by the machine

From ITrees to CakeML

- **We need to show that PureCake semantics are equivalent to CakeML semantics**
- **A different CakeML project already implemented a CESK machine we can use**
- **Turn our interaction trees to CakeML semantics**
    - CakeML uses “oracle semantics”
    - Remember: ITrees simulate *all* possible outside-world semantics
    - We need to carve out the branch from our ITree that corresponds to the CakeML semantics

$$\Delta(e) = r \ \wedge\ k(r) \overset{\Delta}{\rightsquigarrow} tr \ \Rightarrow\ \mathsf{Vis}\ e\ k \overset{\Delta}{\rightsquigarrow} (e, r) :: tr$$

$$\vdash \mathsf{target\_configs\_ok}\ \mathit{config}\ \mathit{machine}\ \wedge\ \mathsf{safe\_itree}\ [\![\mathit{prog}]\!]\ \wedge \\ \mathsf{compile}\ \mathit{config}\ \mathit{prog} = \mathsf{Some}\ \mathit{code}\ \wedge\ \mathsf{code\_in\_memory}\ \mathit{config}\ \mathit{code}\ \mathit{machine} \\ \Rightarrow [\![\mathit{machine}]\!]_{\mathrm{M}}\ \mathsf{prunes}\ [\![\mathit{prog}]\!]$$

Conclusion

The theorems we get from PureCake

- **The compiler compiles *correct* PureCake into *correct* CakeML**
  - That means that if the source code parses and type-checks, it compiles correctly
- **Therefore, it compiles *correct* PureCake into *correct* machine code**

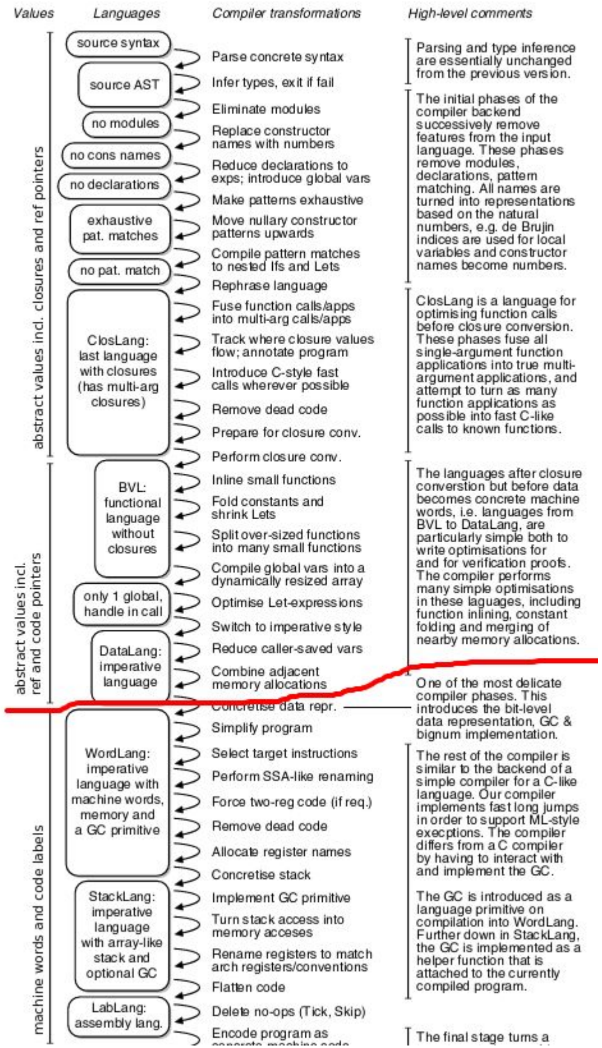
$$
\begin{array}{l}
\vdash \mathsf{compiler}\ str = \mathsf{Some}\ ast_{\text{cake}} \Rightarrow \\
\quad \exists\, ce\ ns.\ \mathsf{frontend}\ str = \mathsf{Some}\ (ce, ns) \land \\
\quad\quad \mathsf{safe\_itree}\ [\![\mathsf{exp\_of}\ ce]\!]_{\mathrm{pure}} \land \\
\quad\quad \mathsf{itree\_rel}\ [\![\mathsf{exp\_of}\ ce]\!]_{\mathrm{pure}}\ [\![ast_{\text{cake}}]\!]_{\text{cake}}
\end{array}
$$

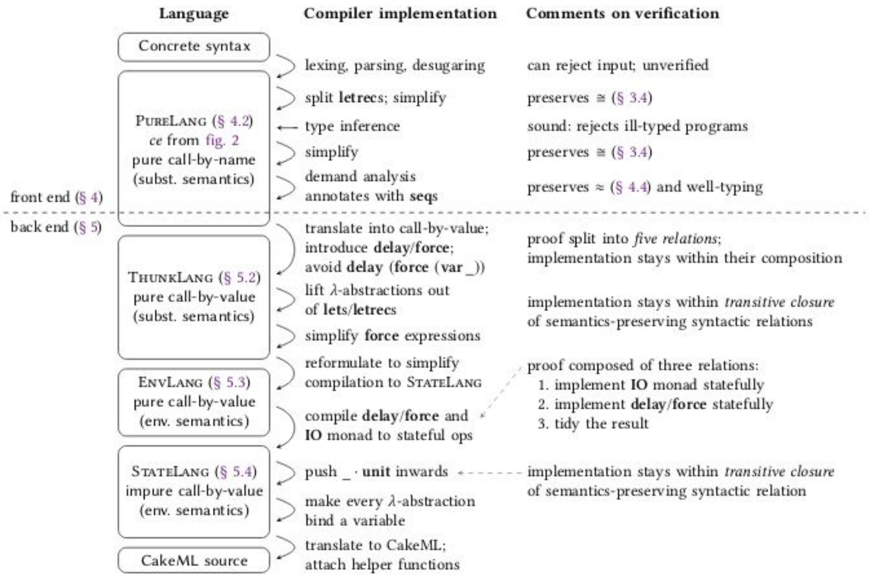
$$
\begin{array}{l}
\vdash \mathsf{frontend}\ str = \mathsf{Some}\ (ce, ns) \Rightarrow \\
\quad \mathsf{safe\_itree}\ [\![\mathsf{exp\_of}\ ce]\!]_{\mathrm{pure}} \land \\
\quad \exists\, ast_{\text{cake}}.\ \mathsf{compiler}\ str = \mathsf{Some}\ ast_{\text{cake}} \land \\
\quad\quad \mathsf{itree\_rel}\ [\![\mathsf{exp\_of}\ ce]\!]_{\mathrm{pure}}\ [\![ast_{\text{cake}}]\!]_{\text{cake}}
\end{array}
$$

$$
\begin{array}{l}
\vdash \mathsf{compiler}\ str = \mathsf{Some}\ ast_{\text{cake}} \land \mathsf{compile}_{\text{cake}}\ config\ ast_{\text{cake}} = \mathsf{Some}\ code \land \\
\quad \mathsf{target\_configs\_ok}\ config\ machine \land \mathsf{code\_in\_memory}\ config\ code\ machine \\
\quad \Rightarrow \exists\, ce\ ns.\ \mathsf{frontend}\ str = \mathsf{Some}\ (ce, ns) \land [\![machine]\!]_{\mathrm{M}}\ \mathsf{prunes}\ [\![\mathsf{exp\_of}\ ce]\!]
\end{array}
$$

The End!

Conclusion





Q and A

“Why no proofs”

- **The proofs are very big**
  - CakeML supposedly takes 22 hours and 16GB of ram to compile/bootstrap from source
- **The proofs are more work than ideas**

“What is Co-inductivity?”

- **Dual to inductive types**
    - Are generated using co-recursive functions
    - Can be potentially infinite
    - Cannot just be consumed by an inductive function

Demand Analysis (vs Haskell)

```haskell
collatzSequence :: Integer -> Integer
collatzSequence n =
  let seqAux acc n =
          if n < 1 then (0-1)
          else if n == 1 then acc
          else seqAux (acc + 1) (collatz n)
  in seqAux 0 n
```

Weak-Head Normal Form

Evaluate the expression until we're stuck at an incomplete lambda or an uninterpretable function

- **Normal Form: We cannot further evaluate the expression**
  - We've evaluated every lambda body as far as we can
  - Basically symbolic evaluation
- **Head Normal Form: We cannot find any lambdas to fill**
  - We've evaluated any top level function bodies
  - (We don't really deal with HNF anymore)
- **Weak Head Normal Form: We can't do trivial substitutions anymore**
  - Any partially applied function will be substituted with its definition any the arguments that were applied
  - We don't do anything else.