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

```
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 CakeML

- **Formally verified compiler for a dialect of Standard ML**

```
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 CakeML

- **Formally verified compiler for a dialect of Standard ML**
- **Takes resource constraints into account for it's proof of correct compilation**

```
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 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**

```
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 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

```
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 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

About this presentation

- **These papers are about a lot of *stuff***

About this presentation

- **These papers are about a lot of *stuff***
  - CakeML and PureCake are big projects

About this presentation

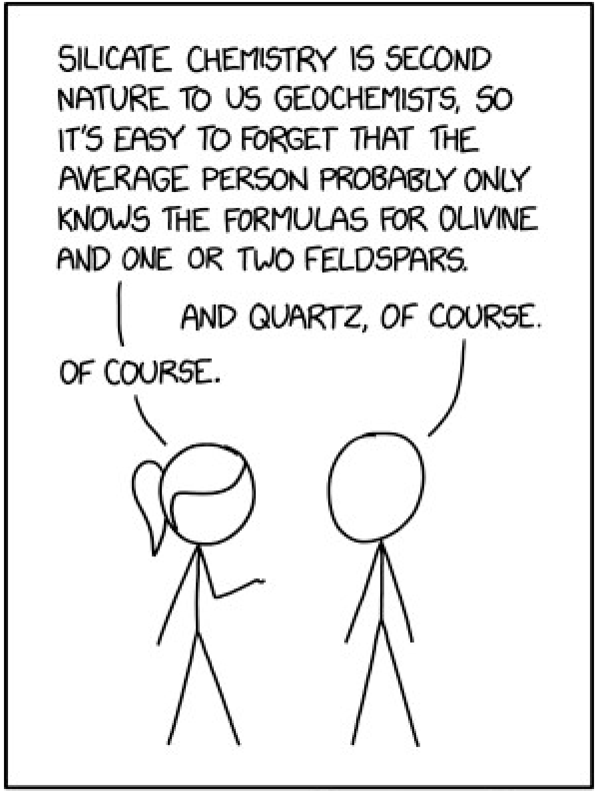
- **These papers are about a lot of *stuff***
  - CakeML and PureCake are big projects
  - A lot of compiler techniques are used to get to where we are

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...**

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…**



EVEN WHEN THEY'RE TRYING TO COMPENSATE FOR IT, EXPERTS IN ANYTHING WILDLY OVERESTIMATE THE AVERAGE PERSON'S FAMILIARITY WITH THEIR FIELD.

How to make a compiler

How to make a compiler

- **Figure out what your starting (source) language does**

How to make a compiler

- **Figure out what your starting (source) language does**
  - What makes a program in the starting language correct?

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?

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**

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 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?

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**

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*

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

I/O effects

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

$$\begin{aligned}
\mathtt{behaviour} &= \mathsf{Diverge}\ (\mathtt{io_event\ stream}) \mid \mathsf{Terminate}\ \mathtt{outcome}\ (\mathtt{io_event\ list}) \mid \mathsf{Fail} \\
\mathtt{outcome} &= \mathsf{Success} \mid \mathsf{Resource_limit_hit} \mid \mathsf{FFI_outcome}\ \mathtt{final_event}
\end{aligned}$$

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**

$$\begin{aligned}
&\vdash \mathsf{compile}\ \mathit{config}\ \mathit{prog} = \mathit{new_prog}\ \wedge \\
&\quad \mathsf{syntactic_condition}\ \mathit{prog}\ \wedge \\
&\quad \mathsf{Fail} \notin \mathsf{semantics}_{\mathrm{A}}\ \mathit{ffi}\ \mathit{prog} \Rightarrow \\
&\quad\ \mathsf{semantics}_{\mathrm{B}}\ \mathit{ffi}\ \mathit{new_prog} = \mathsf{semantics}_{\mathrm{A}}\ \mathit{ffi}\ \mathit{prog}
\end{aligned}$$

General Compiler proofs

The program may run out of memory:

$$\mathsf{semantics}_{\mathrm{B}}\ \mathit{ffi}\ \mathit{new\_prog} \subseteq \mathsf{extend\_with\_resource\_limit}\ (\mathsf{semantics}_{\mathrm{A}}\ \mathit{ffi}\ \mathit{prog})$$

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

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

- **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

```
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
```

```
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
  $([\mathsf{CodePtr}\ ptr;\ \mathsf{Number}\ arg\_count] +\!\!\!+ \mathit{free\_var\_vals})$
- **'RecClosure' ->** Block closure_tag
  $[\mathsf{CodePtr}\ ptr;\ \mathsf{Number}\ arg\_count;\ \mathsf{RefPtr}\ ref\_ptr]$

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

$$
\begin{aligned}
\texttt{exp} \ = \ & \\
& \mathsf{Var}\ \texttt{num} \\
| \ & \mathsf{If}\ \texttt{exp exp exp} \\
| \ & \mathsf{Let}\ (\texttt{exp list})\ \texttt{exp} \\
| \ & \mathsf{Raise}\ \texttt{exp} \\
| \ & \mathsf{Handle}\ \texttt{exp exp} \\
| \ & \mathsf{Tick}\ \texttt{exp} \\
| \ & \mathsf{Call}\ \texttt{num}\ (\texttt{num option})\ (\texttt{exp list}) \\
| \ & \mathsf{Op}\ \texttt{op}\ (\texttt{exp list})
\end{aligned}
$$

```
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

$$
\begin{aligned}
\texttt{prog} =\ & \\
& \mathsf{Skip} \\
\mid\ & \mathsf{Move}\ \texttt{num}\ \texttt{num} \\
\mid\ & \mathsf{Call}\ ((\texttt{num} \times \texttt{num\_set})\ \texttt{option})\ (\texttt{num}\ \texttt{option}) \\
& \quad (\texttt{num}\ \texttt{list})\ ((\texttt{num} \times \texttt{prog})\ \texttt{option}) \\
\mid\ & \mathsf{Assign}\ \texttt{num}\ \texttt{op}\ (\texttt{num}\ \texttt{list})\ (\texttt{num\_set}\ \texttt{option}) \\
\mid\ & \mathsf{Seq}\ \texttt{prog}\ \texttt{prog} \\
\mid\ & \mathsf{If}\ \texttt{num}\ \texttt{prog}\ \texttt{prog} \\
\mid\ & \mathsf{MakeSpace}\ \texttt{num}\ \texttt{num\_set} \\
\mid\ & \mathsf{Raise}\ \texttt{num} \\
\mid\ & \mathsf{Return}\ \texttt{num} \\
\mid\ & \mathsf{Tick}
\end{aligned}
$$

DATALang

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

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

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

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

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

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	ANF

EXP ::= λ VAR . EXP
       | EXP EXP
       | VAR
       | CONST
       | let VAR = EXP in EXP
 	
CONST ::= f | g | h
	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
```

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

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

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

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

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

About PureCake

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

$$\begin{aligned}\mathsf{expand}_x\;[cname[\,\overline{y_n}\,] \to ce',\,\overline{row_m}\,] &\stackrel{\text{def}}{=} \mathbf{if}\;(\mathbf{eq}_?\; cname\; n\,(\mathbf{var}\; x))\;\mathbf{then}\\ &\qquad \mathbf{let}\;\overline{y_n = \mathbf{proj}_n\; cname\;(\mathbf{var}\; x)}\;\mathbf{in}\;(\mathsf{exp\_of}\; ce')\\ \mathsf{expand}_x\;[\,] \stackrel{\text{def}}{=} \mathbf{fail} \qquad &\quad\;\; \mathbf{else}\;\mathsf{expand}_x\;[\,\overline{row_m}\,]\end{aligned}$$

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

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

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

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**

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**

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

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**

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

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

Explained questions

Continuations

Continuations

All programs have a past, a present and a future

Continuations

All programs have a past, a present and a future

- **The past:**

Continuations

All programs have a past, a present and a future

- **The past:**
  - Variables

Continuations

All programs have a past, a present and a future

- **The past:**
    - Variables
    - Assigned memory

Continuations

All programs have a past, a present and a future

- **The past:**
  - Variables
  - Assigned memory
- **The present**

Continuations

All programs have a past, a present and a future

- **The past:**
  - Variables
  - Assigned memory
- **The present**
  - The expression we’re currently evaluating

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

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**

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

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

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)

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

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

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

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

I/O Effects (Interaction trees)

$$(\!| \textbf{diverge},\ \kappa,\ \sigma |\!) = \mathsf{Div} \qquad\qquad (\!| \textbf{error},\ \kappa,\ \sigma |\!) = \mathsf{Ret}\ \textbf{error}$$

$$(\!| \textbf{bind}\ e_1\ e_2,\ \kappa,\ \sigma |\!) = (\!| \mathsf{eval}_{\text{wh}}\ e_1,\ \textbf{bind}\ \bullet\ e_2 :: \kappa,\ \sigma |\!)$$

$$(\!| \textbf{return}\ e,\ \varepsilon,\ \sigma |\!) = \mathsf{Ret}\ \textbf{termination}$$

$$(\!| \textbf{return}\ e_1,\ \textbf{bind}\ \bullet\ e_2 :: \kappa,\ \sigma |\!) = (\!| \mathsf{eval}_{\text{wh}}\ (e_2 \cdot e_1),\ \kappa,\ \sigma |\!)$$

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

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

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

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’**
- **<u>Demand analysis says nothing about evaluation order</u>**

```
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

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

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

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

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

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

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

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

Demand Analysis

Demand Analysis

- **Things get tricky when analysing functions & function calls though**
- **Three cases:**

Demand Analysis

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

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

Demand Analysis

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

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

  - Function arguments need to be demanded when they are applied

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

Demand Analysis

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

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

  - 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

Demand Analysis

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

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

  - Function arguments need to be demanded when they are applied

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

  - The recursive case

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

Compiling PureCake

Parsing

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

Parsing

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

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

Parsing

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

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

- We can now calculate the “Indentation sets of non-terminals

Parsing

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

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

- **We can now calculate the “Indentation sets of non-terminals**
  - Either a closed set of possible indentations (*i* to *j* no. of indents)

Parsing

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

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

- **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)

Parsing

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

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

- **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

Parsing

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

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

- **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)

Parsing

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

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

- **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**

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

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

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

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

Type Inference

Type Inference

- **We now have constraints**

Type Inference

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

Demand Analysis

Demand Analysis

- **We’ve already done this**

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**

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

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

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

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

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

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

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

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

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

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

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

Purecake compilation

StateLang

StateLang

- **We mostly need to prove that our operations on thunks are correct**

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’

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:**

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’

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**

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

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

From ITrees to CakeML

- **We need to show that PureCake semantics are equivalent to CakeML semantics**

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

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**

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”

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

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

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

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

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

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

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

Conclusion

The theorems we get from PureCake

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

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

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

The theorems we get from PureCake

- **The compiler compiles *correct* PureCake into *correct* CakeML**

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

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

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

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

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

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

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

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**

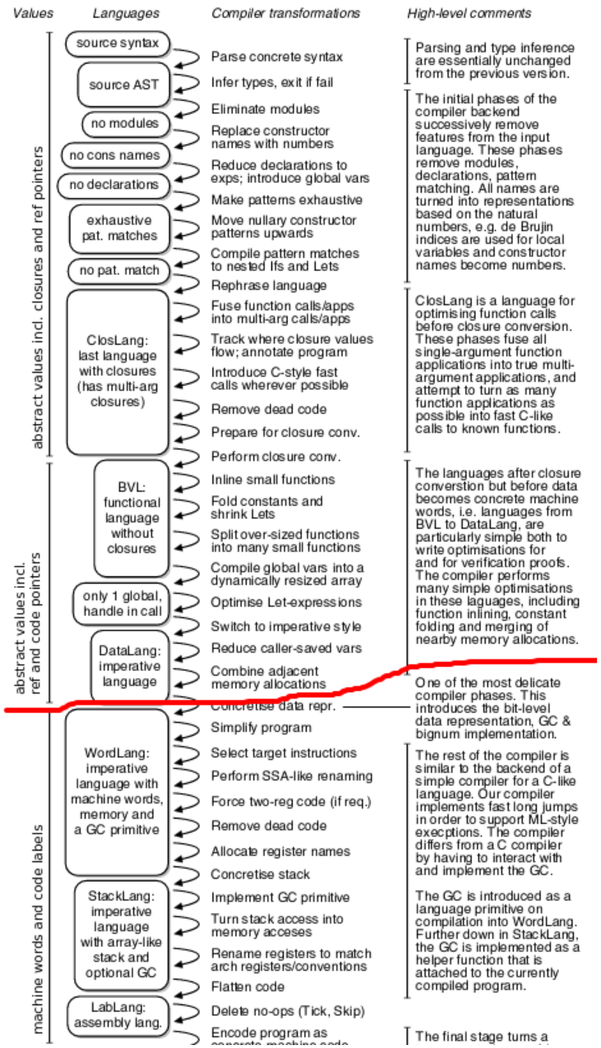
$$\begin{array}{l} \vdash \mathsf{compiler}\ \mathit{str} \ = \ \mathsf{Some}\ \mathit{ast}_{🍰} \ \Rightarrow \\ \quad \exists\, \mathit{ce}\ \mathit{ns}.\ \ \mathsf{frontend}\ \mathit{str} \ = \ \mathsf{Some}\ (\mathit{ce},\ \mathit{ns}) \ \wedge \\ \quad\quad \mathsf{safe\_itree}\ \ [\![\, \mathsf{exp\_of}\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

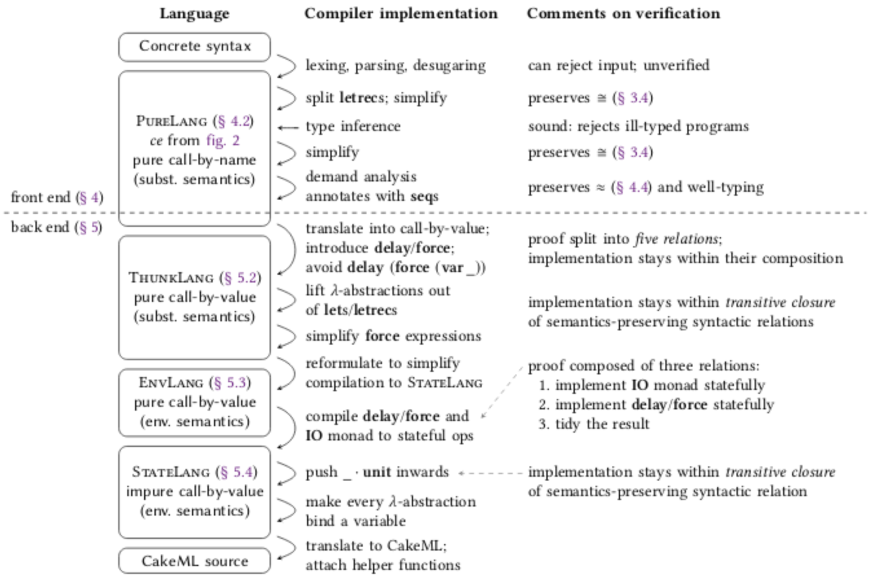
$$\begin{array}{l} \vdash \mathsf{frontend}\ \mathit{str} \ = \ \mathsf{Some}\ (\mathit{ce},\ \mathit{ns}) \ \Rightarrow \\ \quad \mathsf{safe\_itree}\ \ [\![\, \mathsf{exp\_of}\ \mathit{ce} \,]\!]_{\mathrm{pure}} \ \wedge \\ \quad \exists\, \mathit{ast}_{🍰}.\ \ \mathsf{compiler}\ \mathit{str} \ = \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

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

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

Demand Analysis (vs 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
```

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

Weak-Head Normal Form

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

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**

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

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

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**

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

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

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**

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

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.