Automatically Introducing Tail Recursion in CakeML

TFP 2017: The 18th Symposium on Trends in Functional Programming

Oskar Abrahamsson,
Chalmers University of Technology

Tail recursion

Recursive length

```
fun length []       = 0
  | length (x::xs) = length xs + 1
```

Recursive length

```
fun length []        = 0
  | length (x::xs) = length xs + 1
```

+ in tail position

Tail-recursive length

```
fun length’ acc []       = acc
  | length’ acc (x::xs) = length’ (1 + acc) xs
```

```
fun length xs = length’ 0 xs
```

Tail-recursive length

```
fun length' acc []       = acc
  | length' acc (x::xs) = length' (1 + acc) xs
```

Call in tail position

```
fun length xs = length' 0 xs
```

Benefits

- May use tail call elimination:
  - O(1) stack consumption over O(#recursions)
  - Callee re-uses arguments

Benefits

- May use tail call elimination:
  - O(1) stack consumption over O(#recursions)
  - Callee re-uses arguments

➡ Allows for potentially ‘unbounded’ recursion

Rewriting functions using accumulators

Rewriting functions using accumulators

```
fun length []       = 0
  | length (x::xs) = length xs + 1
```

We can do this automatically!

```
fun length' acc []      = acc
  | length' acc (x::xs) = length' (1 + acc) xs

fun length xs = length' 0 xs
```

Rewriting functions using accumulators

If a function `f` has a tail position

```
f x = ... f y + z
```

for some `y, z`

Rewriting functions using accumulators

If a function `f` has a tail position

```
f x = ... f y + z
```

for some `y, z`

introduce an auxiliary function `f’` such that

```
f’ x y = f x + y
```

Procedure

- `f'` is an unused function name
- `a` is the new (accumulating) argument

Procedure

- `f'` is an unused function name
- `a` is the new (accumulating) argument

1. Replace tail expressions `f x + y` by `f' x (y + a)`

Procedure

- `f′` is an unused function name
- `a` is the new (accumulating) argument

1. Replace tail expressions `f x + y` by `f′ x (y + a)`
2. Other tail expressions `e` are replaced by `e + a`

Procedure

- `f′` is an unused function name
- `a` is the new (accumulating) argument

1. Replace tail expressions `f x + y` by `f′ x (y + a)`
2. Other tail expressions `e` are replaced by `e + a`
3. Rename `f` to `f′` and introduce an auxiliary definition in place of `f` s.t. `f x = f′ x 0`

Procedure

- `f′` is an unused function name
- `a` is the new (accumulating) argument

1. Replace tail expressions `f x + y` by `f′ x (y + a)`
2. Other tail expressions `e` are replaced by `e + a`
3. Rename `f` to `f′` and introduce an auxiliary definition in place of `f` s.t. `f x = f′ x 0`

- Works for any associative operator `+` with identity `0`

The CakeML language and compiler

CakeML

- Strongly typed
- Call-by-value semantics
- Based on Standard ML:
  - references, exceptions, modules, I/O, etc…
- Multiple compiler backends:
  - x86-64, 32-bit ARM, 64-bit ARM, MIPS-64, RISC-V

The CakeML compiler

- Implemented in higher-order logic in HOL4.
- Verified to preserve observational semantics:
    - all the way down to machine-code

Target IL for transformation: BVI

(Bytecode-Value Intermediate Language)

- First-order functional language with exceptions
- Functional big-step semantics
- Typeless representation

Reasons for BVI:

- No closures: Can use equality to compare values
- Pass that compiles into BVI provides information about unused ‘function names’.

Introducing tail recursion in BVI

- BVI programs stored in an immutable code store as entries `(nm: num, ar: num, exp: expr)`

Introducing tail recursion in BVI

- BVI programs stored in an immutable code store

  as entries `(nm: num, ar: num, exp: expr)`

Function 'name'

Arity

Expression (body)

Introducing tail recursion in BVI

- BVI programs stored in an immutable code store as entries `(nm: num, ar: num, exp: expr)`
- Do static check which looks for recursive calls under binary operations in tail position (addressed by `nm`)

Introducing tail recursion in BVI

- BVI programs stored in an immutable code store as entries `(nm: num, ar: num, exp: expr)`
- Do static check which looks for recursive calls under binary operations in tail position (addressed by `nm`)
- Perform rewrite almost exactly as previously described

Introducing tail recursion in BVI

- BVI programs stored in an immutable code store

  as entries `(nm: num, ar: num, exp: expr)`
- Do static check which looks for recursive calls

  under binary operations in tail position (addressed by `nm`)
- Perform rewrite almost exactly as previously

  described

- ML languages (and BVI) supports exceptions, mutable state, etc.

- ML languages (and BVI) supports exceptions, mutable state, etc.
- The transformation changes order of evaluation.

- ML languages (and BVI) supports exceptions, mutable state, etc.
- The transformation changes order of evaluation.
- Expressions that are ‘moved up’ must not access global state.

- ML languages (and BVI) supports exceptions, mutable state, etc.
- The transformation changes order of evaluation.
- Expressions that are ‘moved up’ must not access global state.

➡ We perform a pessimistic static check and avoid transforming state-accessing expressions.

Verification of semantic preservation

Top-level theorem:

$$
\begin{aligned}
&\vdash \mathsf{every}\ (\mathsf{free\_names}\ n \circ \mathsf{fst})\ \mathit{prog} \wedge \mathsf{all\_distinct}\ (\mathsf{map}\ \mathsf{fst}\ \mathit{prog}) \wedge \\
&\quad \mathsf{compile}\ n\ \mathit{prog} = \mathit{prog}_2 \wedge \\
&\quad \mathsf{semantics}\ \mathit{ffi}\ (\mathsf{fromAList}\ \mathit{prog})\ \mathit{start} \neq \mathsf{Fail} \Rightarrow \\
&\qquad \mathsf{semantics}\ \mathit{ffi}\ (\mathsf{fromAList}\ \mathit{prog})\ \mathit{start} = \\
&\qquad\quad \mathsf{semantics}\ \mathit{ffi}\ (\mathsf{fromAList}\ \mathit{prog}_2)\ \mathit{start}
\end{aligned}
$$

Top-level theorem:

$$
\begin{array}{l}
\vdash \mathsf{every}\ (\mathsf{free\_names}\ n \circ \mathsf{fst})\ \mathit{prog} \land \mathsf{all\_distinct}\ (\mathsf{map\ fst}\ \mathit{prog}) \land \\
\quad \mathsf{compile}\ n\ \mathit{prog} = \mathit{prog}_2 \land \\
\quad \boxed{\begin{array}{l}\mathsf{semantics}\ \mathit{ffi}\ (\mathsf{fromAList}\ \mathit{prog})\ \mathit{start} \neq \mathsf{Fail} \Rightarrow \\
\quad \mathsf{semantics}\ \mathit{ffi}\ (\mathsf{fromAList}\ \mathit{prog})\ \mathit{start} = \\
\quad\quad \mathsf{semantics}\ \mathit{ffi}\ (\mathsf{fromAList}\ \mathit{prog}_2)\ \mathit{start}\end{array}}
\end{array}
$$

original program was valid

$\Longrightarrow$

semantics are preserved under the transformation `compile`

- `semantics` describes the semantics of programs
- Defined in terms of a function `evaluate`:
  - describes the semantics of *expressions*

evaluate

$$
\begin{array}{l}
\textsf{evaluate}\ ([],\mathit{env},s) = (\textsf{Rval}\ [],s) \\[4pt]
\textsf{evaluate}\ (x::y::\mathit{xs},\mathit{env},s) = \\
\quad \textsf{case evaluate}\ ([x],\mathit{env},s)\ \textsf{of} \\
\qquad (\textsf{Rval}\ v_1,s_1)\ \Rightarrow \\
\qquad\quad (\textsf{case evaluate}\ (y::\mathit{xs},\mathit{env},s_1)\ \textsf{of} \\
\qquad\qquad (\textsf{Rval}\ \mathit{vs},s_2)\ \Rightarrow\ (\textsf{Rval}\ (\textsf{hd}\ v_1::\mathit{vs}),s_2) \\
\qquad\quad\ \mid (\textsf{Rerr}\ v_8,s_2)\ \Rightarrow\ (\textsf{Rerr}\ v_8,s_2)) \\
\quad \mid (\textsf{Rerr}\ v_8,s_1)\ \Rightarrow\ (\textsf{Rerr}\ v_8,s_1) \\[4pt]
\ldots \\[4pt]
\textsf{evaluate}\ ([\textsf{Op}\ \mathit{op}\ \mathit{xs}],\mathit{env},s) = \\
\quad \textsf{case evaluate}\ (\mathit{xs},\mathit{env},s)\ \textsf{of} \\
\qquad (\textsf{Rval}\ \mathit{vs},s')\ \Rightarrow \\
\qquad\quad (\textsf{case do\_app}\ \mathit{op}\ (\textsf{reverse}\ \mathit{vs})\ s'\ \textsf{of} \\
\qquad\qquad \textsf{Rval}\ (v,s')\ \Rightarrow\ (\textsf{Rval}\ [v],s') \\
\qquad\quad\ \mid \textsf{Rerr}\ e\ \Rightarrow\ (\textsf{Rerr}\ e,s')) \\
\quad \mid (\textsf{Rerr}\ v_7,s')\ \Rightarrow\ (\textsf{Rerr}\ v_7,s')
\end{array}
$$

$$
\begin{array}{l}
\vdash \mathsf{evaluate}\ (xs,env_1,s) = (r,t) \land \\
\quad \mathsf{env\_rel}\ transformed\ acc\ env_1\ env_2 \land \mathsf{code\_rel}\ s.\mathsf{code}\ c \land \\
\quad (transformed \Rightarrow \mathsf{length}\ xs = 1) \land \\
\quad r \neq \mathsf{Rerr}\ (\mathsf{Rabort}\ \mathsf{Rtype\_error}) \Rightarrow \\
\quad\ \ \mathsf{evaluate}\ (xs,env_2,s\ \mathsf{with\ code}\ :=\ c) = \\
\quad\quad\ (r,t\ \mathsf{with\ code}\ :=\ c) \land \\
\quad\ \ (transformed \Rightarrow \\
\quad\quad\ \forall\, op\ n\ exp\ ar. \\
\quad\quad\quad \mathsf{lookup}\ nm\ s.\mathsf{code} = \mathsf{Some}\ (ar,exp) \land \\
\quad\quad\quad \mathsf{is\_transformed\_code}\ nm\ ar\ exp\ n\ c\ op \land \\
\quad\quad\quad \mathsf{tail\_is\_ok}\ nm\ (\mathsf{hd}\ xs) = \mathsf{Some}\ op \Rightarrow \\
\quad\quad\quad\ \ \mathsf{evaluate} \\
\quad\quad\quad\quad ([\mathsf{transform\_tail}\ n\ op\ nm\ acc\ (\mathsf{hd}\ xs)],env_2, \\
\quad\quad\quad\quad\ s\ \mathsf{with\ code}\ :=\ c) = \\
\quad\quad\quad\quad \mathsf{evaluate} \\
\quad\quad\quad\quad\ \ ([\mathsf{apply\_op}\ op\ (\mathsf{hd}\ xs)\ (\mathsf{Var}\ acc)], \\
\quad\quad\quad\quad\quad env_2,s\ \mathsf{with\ code}\ :=\ c))
\end{array}
$$

Semantics preservation theorem for expressions

$$
\begin{array}{l}
\vdash \mathsf{evaluate}\ (xs, env_1, s) = (r, t) \wedge \\
\quad \boxed{\mathsf{env\_rel}\ transformed\ acc\ env_1\ env_2 \wedge \mathsf{code\_rel}\ s.\mathsf{code}\ c \wedge} \\
\quad (transformed \Rightarrow \mathsf{length}\ xs = 1) \wedge \\
\quad r \neq \mathsf{Rerr}\ (\mathsf{Rabort}\ \mathsf{Rtype\_error}) \Rightarrow \\
\quad\quad \mathsf{evaluate}\ (xs, env_2, s\ \mathsf{with}\ \mathsf{code}\ := c) = \\
\quad\quad\quad (r, t\ \mathsf{with}\ \mathsf{code}\ := c) \wedge \\
\quad\quad (transformed \Rightarrow \\
\quad\quad\quad \forall\, op\ n\ exp\ ar. \\
\quad\quad\quad\quad \mathsf{lookup}\ nm\ s.\mathsf{code} = \mathsf{Some}\ (ar, exp) \wedge \\
\quad\quad\quad\quad \mathsf{is\_transformed\_code}\ nm\ ar\ exp\ n\ c\ op \wedge \\
\quad\quad\quad\quad \mathsf{tail\_is\_ok}\ nm\ (\mathsf{hd}\ xs) = \mathsf{Some}\ op \Rightarrow \\
\quad\quad\quad\quad\quad \mathsf{evaluate} \\
\quad\quad\quad\quad\quad\quad ([\mathsf{transform\_tail}\ n\ op\ nm\ acc\ (\mathsf{hd}\ xs)], env_2, \\
\quad\quad\quad\quad\quad\quad\ s\ \mathsf{with}\ \mathsf{code}\ := c) = \\
\quad\quad\quad\quad\quad\quad \mathsf{evaluate} \\
\quad\quad\quad\quad\quad\quad\quad ([\mathsf{apply\_op}\ op\ (\mathsf{hd}\ xs)\ (\mathsf{Var}\ acc)], \\
\quad\quad\quad\quad\quad\quad\quad\ env_2, s\ \mathsf{with}\ \mathsf{code}\ := c))
\end{array}
$$

Accumulator points to ‘right’ place in environment,
and c is the transformed code store…

$$
\begin{array}{l}
\vdash \boxed{\mathsf{evaluate}\ (xs, env_1, s) = (r, t)\ \wedge} \\
\quad \mathsf{env\_rel}\ transformed\ acc\ env_1\ env_2 \wedge \mathsf{code\_rel}\ s.\mathsf{code}\ c\ \wedge \\
\quad (transformed \Rightarrow \mathsf{length}\ xs = 1)\ \wedge \\
\quad \boxed{r \neq \mathsf{Rerr}\ (\mathsf{Rabort}\ \mathsf{Rtype\_error}) \Rightarrow} \\
\qquad \mathsf{evaluate}\ (xs, env_2, s\ \mathsf{with\ code} := c) = \\
\qquad\quad (r, t\ \mathsf{with\ code} := c)\ \wedge \\
\qquad (transformed \Rightarrow \\
\qquad\quad \forall\, op\ n\ exp\ ar. \\
\qquad\qquad \mathsf{lookup}\ nm\ s.\mathsf{code} = \mathsf{Some}\ (ar, exp)\ \wedge \\
\qquad\qquad \mathsf{is\_transformed\_code}\ nm\ ar\ exp\ n\ c\ op\ \wedge \\
\qquad\qquad \mathsf{tail\_is\_ok}\ nm\ (\mathsf{hd}\ xs) = \mathsf{Some}\ op \Rightarrow \\
\qquad\qquad\quad \mathsf{evaluate} \\
\qquad\qquad\qquad ([\mathsf{transform\_tail}\ n\ op\ nm\ acc\ (\mathsf{hd}\ xs)], env_2, \\
\qquad\qquad\qquad\ s\ \mathsf{with\ code} := c) = \\
\qquad\qquad\qquad \mathsf{evaluate} \\
\qquad\qquad\qquad\quad ([\mathsf{apply\_op}\ op\ (\mathsf{hd}\ xs)\ (\mathsf{Var}\ acc)], \\
\qquad\qquad\qquad\quad\ env_2, s\ \mathsf{with\ code} := c))
\end{array}
$$

…original program does not have semantics `Fail`

$$
\begin{array}{l}
\vdash \mathsf{evaluate}\ (xs,env_1,s) = (r,t) \wedge \\
\quad \mathsf{env\_rel}\ transformed\ acc\ env_1\ env_2 \wedge \mathsf{code\_rel}\ s.\mathsf{code}\ c \wedge \\
\quad (transformed \Rightarrow \mathsf{length}\ xs = 1) \wedge \\
\quad r \neq \mathsf{Rerr}\ (\mathsf{Rabort}\ \mathsf{Rtype\_error}) \Rightarrow \\
\quad \boxed{\begin{array}{l} \mathsf{evaluate}\ (xs,env_2,s\ \mathsf{with}\ \mathsf{code} := c) = \\ \quad (r,t\ \mathsf{with}\ \mathsf{code} := c) \wedge \end{array}} \\
\qquad (transformed \Rightarrow \\
\qquad\quad \forall\ op\ n\ exp\ ar. \\
\qquad\qquad \mathsf{lookup}\ nm\ s.\mathsf{code} = \mathsf{Some}\ (ar,exp) \wedge \\
\qquad\qquad \mathsf{is\_transformed\_code}\ nm\ ar\ exp\ n\ c\ op \wedge \\
\qquad\qquad \mathsf{tail\_is\_ok}\ nm\ (\mathsf{hd}\ xs) = \mathsf{Some}\ op \Rightarrow \\
\qquad\qquad\quad \mathsf{evaluate} \\
\qquad\qquad\qquad ([\mathsf{transform\_tail}\ n\ op\ nm\ acc\ (\mathsf{hd}\ xs)],env_2, \\
\qquad\qquad\qquad\ s\ \mathsf{with}\ \mathsf{code} := c) = \\
\qquad\qquad\qquad \mathsf{evaluate} \\
\qquad\qquad\qquad\quad ([\mathsf{apply\_op}\ op\ (\mathsf{hd}\ xs)\ (\mathsf{Var}\ acc)], \\
\qquad\qquad\qquad\quad\ env_2,s\ \mathsf{with}\ \mathsf{code} := c))
\end{array}
$$

…then semantics are preserved under transformation

$$
\begin{aligned}
&\vdash \mathsf{evaluate}\ (xs,env_1,s) = (r,t)\ \wedge \\
&\quad \mathsf{env\_rel}\ transformed\ acc\ env_1\ env_2 \wedge \mathsf{code\_rel}\ s.\mathsf{code}\ c\ \wedge \\
&\quad (transformed \Rightarrow \mathsf{length}\ xs = 1)\ \wedge \\
&\quad r \neq \mathsf{Rerr}\ (\mathsf{Rabort}\ \mathsf{Rtype\_error}) \Rightarrow \\
&\qquad \mathsf{evaluate}\ (xs,env_2,s\ \mathsf{with\ code}\ := c) = \\
&\qquad\quad (r,t\ \mathsf{with\ code}\ := c)\ \wedge \\
&\qquad \boxed{\begin{aligned}
&(transformed \Rightarrow \\
&\quad \forall\, op\ n\ exp\ ar. \\
&\qquad \mathsf{lookup}\ nm\ s.\mathsf{code} = \mathsf{Some}\ (ar,exp)\ \wedge \\
&\qquad \mathsf{is\_transformed\_code}\ nm\ ar\ exp\ n\ c\ op\ \wedge \\
&\qquad \mathsf{tail\_is\_ok}\ nm\ (\mathsf{hd}\ xs) = \mathsf{Some}\ op \Rightarrow \\
&\qquad\quad \mathsf{evaluate} \\
&\qquad\qquad ([\mathsf{transform\_tail}\ n\ op\ nm\ acc\ (\mathsf{hd}\ xs)],env_2, \\
&\qquad\qquad\ s\ \mathsf{with\ code}\ := c) = \\
&\qquad\qquad \mathsf{evaluate} \\
&\qquad\qquad\quad ([\mathsf{apply\_op}\ op\ (\mathsf{hd}\ xs)\ (\mathsf{Var}\ acc)], \\
&\qquad\qquad\quad\ env_2,s\ \mathsf{with\ code}\ := c))
\end{aligned}}
\end{aligned}
$$

… and if `exp` was transformed into `exp'`, then

`exp' ≡ exp + acc`

- Transformation is implemented as stand-alone compiler pass in the CakeML compiler

- Transformation is implemented as stand-alone compiler pass in the CakeML compiler

- Fully verified

- Transformation is implemented as stand-alone compiler pass in the CakeML compiler
- Fully verified
- Supports associative integer arithmetic

- Transformation is implemented as stand-alone compiler pass in the CakeML compiler
- Fully verified
- Supports associative integer arithmetic
- Implementation exists for list-append (not yet verified)

Syntactic conditions

Op goal

Tail position with recursive call under operation:

```
evaluate (f xs ++ ys) ≠ Fail
...
⟹
evaluate (f xs ++ ys) =
  evaluate (f′ xs ys)
```

Order of evaluation

`evaluate (f xs ++ ys)`	`evaluate (f′ xs ys)`
`xs`	`xs`
Body of `f`	`ys`
`ys`	Body of `f′` (‘same’ as `f`)
`do_app ...`	`...`

Order of evaluation

`evaluate (f xs ++ ys)`	`evaluate (f' xs ys)`
`xs`	`xs`
Body of `f`	`ys`
`ys`	Body of `f'` (‘same’ as `f`)
`do_app ...`	`...`

Challenge:

```
evaluate (f xs ++ ys) ≠ Fail
...
```

$$\Longrightarrow$$

```
evaluate (f xs ++ ys) =
evaluate (f′ xs ys)
```

Challenge:

What if `evaluate (ys) = Fail`?

$$\texttt{evaluate (f xs ++ ys)} \neq \texttt{Fail}$$

$$\vdots$$

$$\Longrightarrow$$

```
evaluate (f xs ++ ys) =
  evaluate (f′ xs ys)
```

Challenge:

What if `evaluate (ys) = Fail`?

- `ys` does not access global state

Challenge:

What if `evaluate (ys) = Fail`?

- `ys` does not access global state
- `ys` is pure

Challenge:

What if `evaluate (ys) = Fail`?

- `ys` does not access global state
- `ys` is pure
- `ys` will evaluate to a value

Challenge:

What if `evaluate (ys) = Fail`?

- `ys` does not access global state
- `ys` is pure
- `ys` will evaluate to a value
- this value will be of correct type

Challenge:

What if `evaluate (ys) = Fail`?

- `ys` does not access global state
- `ys` is pure
- `ys` will evaluate to a value
- this value will be of correct type

➡ `evaluate (ys) ≠ Fail`

Challenge:

What if `evaluate (ys) = Fail`?

- `ys` does not access global state
- `ys` is pure
- `ys` will evaluate to a value
- this value will be of correct type

➡ ~~`evaluate (ys) ≠ Fail`~~

However, this is not the case

Consider the following program:

```
exception Foo;
fun foo (x: int) = (raise Foo) + x
```

Consider the following program:

```
exception Foo;
fun foo (x: int) = (raise Foo) + x
```

Equivalent BVI expression:

```
Raise Foo + x
```

What if…

`Raise Foo + x`   `x + Raise Foo`

Valid:

- LHS does not evaluate to `Rerr Fail`
- `x:int` is pure and `+` is commutative.

What if…

`Raise Foo + x` `x + Raise Foo`

Valid:

- LHS does not evaluate to `Rerr Fail`
- `x:int` is pure and `+` is commutative.

RHS should still evaluate to `Rerr Foo`

$$\frac{(\Gamma,\texttt{s}) \vdash \texttt{Raise Foo} \Downarrow \texttt{Rerr Foo} \qquad (\Gamma,\texttt{s}) \vdash \texttt{x} \Downarrow \texttt{?}}{(\Gamma,\texttt{s}) \vdash \texttt{Raise Foo + x} \Downarrow \texttt{Rerr Foo}}$$

$$\frac{(\Gamma,\mathtt{s}) \vdash \mathtt{Raise\ Foo} \downarrow \mathtt{Rerr\ Foo} \qquad (\Gamma,\mathtt{s}) \vdash \mathtt{x} \downarrow \mathtt{?}}{(\Gamma,\mathtt{s}) \vdash \mathtt{Raise\ Foo + x} \downarrow \mathtt{Rerr\ Foo}}$$

We can deduce nothing about `x`!

$$\frac{(\Gamma, \mathtt{s}) \vdash \mathtt{Raise\ Foo} \downarrow \mathtt{Rerr\ Foo} \qquad (\Gamma, \mathtt{s}) \vdash \mathtt{x} \downarrow\ ?}{(\Gamma, \mathtt{s}) \vdash \mathtt{Raise\ Foo\ +\ x} \downarrow \mathtt{Rerr\ Foo}}$$

- `x` can be an unbound variable
- `x` can point to a value of any type

$$\frac{(\Gamma, \mathtt{s}) \vdash \mathtt{Raise\ Foo} \downarrow \mathtt{Rerr\ Foo} \qquad (\Gamma, \mathtt{s}) \vdash \mathtt{x} \downarrow \mathtt{?}}{(\Gamma, \mathtt{s}) \vdash \mathtt{Raise\ Foo + x} \downarrow \mathtt{Rerr\ Foo}}$$

- `x` can be an unbound variable
- `x` can point to a value of any type

`Raise Foo + x` is still ‘valid’

Assumption

$$\frac{}{(\Gamma, s) \vdash x \Downarrow \mathtt{Rerr\ Fail}}$$

$$\frac{}{(\Gamma,\mathtt{s}) \vdash \mathtt{x} \Downarrow \mathtt{Rerr\ Fail}}$$

$$\frac{(\Gamma,\mathtt{s}) \vdash \mathtt{Raise\ Foo} \Downarrow \mathtt{Rerr\ Foo} \qquad (\Gamma,\mathtt{s}) \vdash \mathtt{x} \Downarrow \mathtt{Rerr\ Fail}}{(\Gamma,\mathtt{s}) \vdash \mathtt{Raise\ Foo} + \mathtt{x} \Downarrow \mathtt{Rerr\ Foo}}$$

$$\frac{(\Gamma,\mathtt{s}) \vdash \mathtt{x} \Downarrow \mathtt{Rerr\ Fail} \qquad (\Gamma,\mathtt{s}) \vdash \mathtt{Raise\ Foo} \Downarrow \mathtt{Rerr\ Foo}}{(\Gamma,\mathtt{s}) \vdash \mathtt{x} + \mathtt{Raise\ Foo} \Downarrow \mathtt{Rerr\ Fail}}$$

Conclusion

`Raise Foo + x` $\neq$ `x + Raise Foo`

(unless we have stronger guarantees about `x`)

Strong guarantees?

- x does not access global state in any way

Strong guarantees?

- x does not access global state in any way
- x is known at compile-time to evaluate to a well-typed value

Strong guarantees?

- x does not access global state in any way
- x is known at compile-time to evaluate to a well-typed value

➡ x cannot be a variable

Strong guarantees?

- x does not access global state in any way
- x is known at compile-time to evaluate to a well-typed value

➡ x cannot be a variable

Too strong!

Providing guarantees

- Give all (functional) ILs a type system:
  - A lot of work (prove soundness preservation between each stage)

Providing guarantees

- Give all (functional) ILs a type system:
    - A lot of work (prove soundness preservation between each stage)
- Ad-hoc solution:
    - Some expressions can only be valid if variables evaluate to well-typed values

Providing guarantees

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

Providing guarantees

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

n is bound
or this evaluates to
Rerr Fail

Providing guarantees

`n:int` or
this evaluates to
Rerr Fail

`n` is bound
or this evaluates to
Rerr Fail

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

Providing guarantees

`n:int` or
this evaluates to
Rerr Fail

`n` is bound
or this evaluates to
Rerr Fail

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

Enough information
to move `(n*)`

Providing guarantees

`n:int` or
this evaluates to
Rerr Fail

`n` is bound
or this evaluates to
Rerr Fail

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

Enough information
to move `(n*)`

We can build a context (additional environment)
with this information.

Summary

- Implemented a code transformation which introduces tail recursion in the CakeML compiler

- Implemented a code transformation which introduces tail recursion in the CakeML compiler
- Verified that transformation preserves semantics

- Implemented a code transformation which introduces tail recursion in the CakeML compiler
- Verified that transformation preserves semantics
- Discussed challenges with verification technique

- Implemented a code transformation which introduces tail recursion in the CakeML compiler
- Verified that transformation preserves semantics
- Discussed challenges with verification technique
- Suggested workarounds (types/context)

- Implemented a code transformation which introduces tail recursion in the CakeML compiler
- Verified that transformation preserves semantics
- Discussed challenges with verification technique
- Suggested workarounds (types/context)

Thank you for listening!

Questions?

Learn more about CakeML: **https://cakeml.org** GitHub: **https://github.com/CakeML/cakeml**