

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 $O(\text{\#recursions})$
 - Callee re-uses arguments

Benefits

- May use tail call elimination:
 - $O(1)$ stack consumption over $O(\text{\#recursions})$
 - 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

$$f\ x = \dots f\ y + z$$

for some y, z

Rewriting functions using accumulators

If a function f has a tail position

$$f\ x = \dots f\ y + z$$

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:

$\vdash \text{every } (\text{free_names } n \circ \text{fst}) \text{ } prog \wedge \text{all_distinct } (\text{map fst } prog) \wedge$
 $\text{compile } n \text{ } prog = prog_2 \wedge$
 $\text{semantics } ffi \text{ (fromAList } prog) \text{ start} \neq \text{Fail} \Rightarrow$
 $\text{semantics } ffi \text{ (fromAList } prog) \text{ start} =$
 $\text{semantics } ffi \text{ (fromAList } prog_2) \text{ start}$

Top-level theorem:

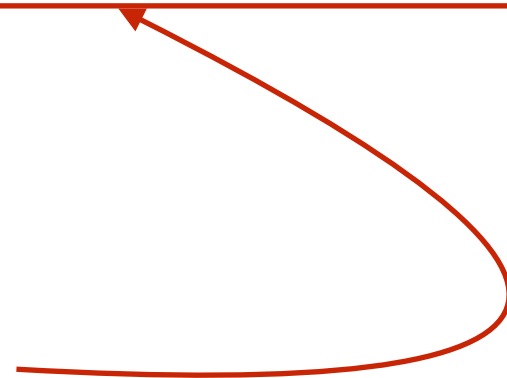
$\vdash \text{every } (\text{free_names } n \circ \text{fst}) \text{ prog} \wedge \text{all_distinct } (\text{map fst prog}) \wedge$
 $\text{compile } n \text{ prog} = \text{prog}_2 \wedge$

$\text{semantics } ffi \text{ (fromAList prog) start} \neq \text{Fail} \Rightarrow$
 $\text{semantics } ffi \text{ (fromAList prog) start} =$
 $\text{semantics } ffi \text{ (fromAList prog}_2\text{) start}$

original program was valid

\Rightarrow

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

$\text{evaluate } ([], env, s) = (\text{Rval } [], s)$

$\text{evaluate } (x::y::xs, env, s) =$
 $\text{case evaluate } ([x], env, s) \text{ of}$
 $(\text{Rval } v_1, s_1) \Rightarrow$
 $(\text{case evaluate } (y::xs, env, s_1) \text{ of}$
 $(\text{Rval } vs, s_2) \Rightarrow (\text{Rval } (\text{hd } v_1::vs), s_2)$
 $| (\text{Rerr } v_8, s_2) \Rightarrow (\text{Rerr } v_8, s_2))$
 $| (\text{Rerr } v_8, s_1) \Rightarrow (\text{Rerr } v_8, s_1)$

...

$\text{evaluate } ([\text{Op } op \ xs], env, s) =$
 $\text{case evaluate } (xs, env, s) \text{ of}$
 $(\text{Rval } vs, s') \Rightarrow$
 $(\text{case do_app } op \ (\text{reverse } vs) \ s' \text{ of}$
 $\text{Rval } (v, s') \Rightarrow (\text{Rval } [v], s')$
 $| \text{Rerr } e \Rightarrow (\text{Rerr } e, s'))$
 $| (\text{Rerr } v_7, s') \Rightarrow (\text{Rerr } v_7, s')$

$$\begin{aligned}
&\vdash \text{evaluate } (xs, env_1, s) = (r, t) \wedge \\
&\quad env_rel \text{ transformed } acc \ env_1 \ env_2 \wedge \text{code_rel } s.code \ c \wedge \\
&\quad (transformed \Rightarrow \text{length } xs = 1) \wedge \\
&\quad r \neq \text{Rerr } (\text{Rabort } \text{Rtype_error}) \Rightarrow \\
&\quad \text{evaluate } (xs, env_2, s \text{ with code } := c) = \\
&\quad \quad (r, t \text{ with code } := c) \wedge \\
&\quad (transformed \Rightarrow \\
&\quad \quad \forall op \ n \ exp \ ar. \\
&\quad \quad \text{lookup } nm \ s.code = \text{Some } (ar, exp) \wedge \\
&\quad \quad \text{is_transformed_code } nm \ ar \ exp \ n \ c \ op \wedge \\
&\quad \quad \text{tail_is_ok } nm \ (\text{hd } xs) = \text{Some } op \Rightarrow \\
&\quad \quad \text{evaluate} \\
&\quad \quad \quad ([\text{transform_tail } n \ op \ nm \ acc \ (\text{hd } xs)], env_2, \\
&\quad \quad \quad s \text{ with code } := c) = \\
&\quad \quad \text{evaluate} \\
&\quad \quad \quad ([\text{apply_op } op \ (\text{hd } xs) \ (\text{Var } acc)], \\
&\quad \quad \quad env_2, s \text{ with code } := c))
\end{aligned}$$

Semantics preservation theorem for expressions

$$\begin{aligned}
&\vdash \text{evaluate } (xs, env_1, s) = (r, t) \wedge \\
&\quad \boxed{\text{env_rel transformed acc env}_1 \text{ env}_2 \wedge \text{code_rel } s.\text{code } c \wedge} \\
&\quad (transformed \Rightarrow \text{length } xs = 1) \wedge \\
&\quad r \neq \text{Rerr (Rabort Rtype_error)} \Rightarrow \\
&\quad \text{evaluate } (xs, env_2, s \text{ with code } := c) = \\
&\quad \quad (r, t \text{ with code } := c) \wedge \\
&\quad (transformed \Rightarrow \\
&\quad \quad \forall op \ n \ exp \ ar. \\
&\quad \quad \text{lookup } nm \ s.\text{code} = \text{Some } (ar, exp) \wedge \\
&\quad \quad \text{is_transformed_code } nm \ ar \ exp \ n \ c \ op \wedge \\
&\quad \quad \text{tail_is_ok } nm \ (\text{hd } xs) = \text{Some } op \Rightarrow \\
&\quad \quad \text{evaluate} \\
&\quad \quad \quad ([\text{transform_tail } n \ op \ nm \ acc \ (\text{hd } xs)], env_2, \\
&\quad \quad \quad \quad s \text{ with code } := c) = \\
&\quad \quad \text{evaluate} \\
&\quad \quad \quad ([\text{apply_op } op \ (\text{hd } xs) \ (\text{Var } acc)], \\
&\quad \quad \quad \quad env_2, s \text{ with code } := c))
\end{aligned}$$

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

$$\begin{aligned}
& \vdash \text{evaluate } (xs, env_1, s) = (r, t) \wedge \\
& \quad env_rel \text{ transformed } acc \ env_1 \ env_2 \wedge code_rel \ s.code \ c \wedge \\
& \quad (transformed \Rightarrow \text{length } xs = 1) \wedge \\
& \quad r \neq \text{Rerr (Rabort Rtype_error)} \Rightarrow \\
& \quad \text{evaluate } (xs, env_2, s \text{ with code } := c) = \\
& \quad \quad (r, t \text{ with code } := c) \wedge \\
& \quad (transformed \Rightarrow \\
& \quad \quad \forall op \ n \ exp \ ar. \\
& \quad \quad \text{lookup } nm \ s.code = \text{Some } (ar, exp) \wedge \\
& \quad \quad \text{is_transformed_code } nm \ ar \ exp \ n \ c \ op \wedge \\
& \quad \quad \text{tail_is_ok } nm \ (\text{hd } xs) = \text{Some } op \Rightarrow \\
& \quad \quad \text{evaluate} \\
& \quad \quad \quad ([\text{transform_tail } n \ op \ nm \ acc \ (\text{hd } xs)], env_2, \\
& \quad \quad \quad \quad s \text{ with code } := c) = \\
& \quad \quad \text{evaluate} \\
& \quad \quad \quad ([\text{apply_op } op \ (\text{hd } xs) \ (\text{Var } acc)], \\
& \quad \quad \quad \quad env_2, s \text{ with code } := c))
\end{aligned}$$

...original program does not have semantics Fail

$$\begin{aligned}
&\vdash \text{evaluate } (xs, env_1, s) = (r, t) \wedge \\
&\quad env_rel \text{ transformed } acc \ env_1 \ env_2 \wedge \text{code_rel } s.code \ c \wedge \\
&\quad (transformed \Rightarrow \text{length } xs = 1) \wedge \\
&\quad r \neq \text{Rerr } (\text{Rabort } \text{Rtype_error}) \Rightarrow \\
&\quad \boxed{\text{evaluate } (xs, env_2, s \text{ with code } := c) =} \\
&\quad \boxed{(r, t \text{ with code } := c) \wedge} \\
&\quad (transformed \Rightarrow \\
&\quad \quad \forall op \ n \ exp \ ar. \\
&\quad \quad \text{lookup } nm \ s.code = \text{Some } (ar, exp) \wedge \\
&\quad \quad \text{is_transformed_code } nm \ ar \ exp \ n \ c \ op \wedge \\
&\quad \quad \text{tail_is_ok } nm \ (\text{hd } xs) = \text{Some } op \Rightarrow \\
&\quad \quad \text{evaluate} \\
&\quad \quad ([\text{transform_tail } n \ op \ nm \ acc \ (\text{hd } xs)], env_2, \\
&\quad \quad \quad s \text{ with code } := c) = \\
&\quad \quad \text{evaluate} \\
&\quad \quad ([\text{apply_op } op \ (\text{hd } xs) \ (\text{Var } acc)], \\
&\quad \quad \quad env_2, s \text{ with code } := c))
\end{aligned}$$

...then semantics are preserved under transformation

$$\vdash \text{evaluate } (xs, env_1, s) = (r, t) \wedge$$

$$\text{env_rel transformed acc env}_1 \text{ env}_2 \wedge \text{code_rel } s.\text{code } c \wedge$$

$$(transformed \Rightarrow \text{length } xs = 1) \wedge$$

$$r \neq \text{Rerr (Rabort Rtype_error)} \Rightarrow$$

$$\text{evaluate } (xs, env_2, s \text{ with code } := c) =$$

$$(r, t \text{ with code } := c) \wedge$$

$$(transformed \Rightarrow$$

$$\forall op \ n \ exp \ ar.$$

$$\text{lookup } nm \ s.\text{code} = \text{Some } (ar, exp) \wedge$$

$$\text{is_transformed_code } nm \ ar \ exp \ n \ c \ op \wedge$$

$$\text{tail_is_ok } nm \ (\text{hd } xs) = \text{Some } op \Rightarrow$$

$$\text{evaluate}$$

$$([\text{transform_tail } n \ op \ nm \ acc \ (\text{hd } xs)], env_2,$$

$$s \text{ with code } := c) =$$

$$\text{evaluate}$$

$$([\text{apply_op } op \ (\text{hd } xs) \ (\text{Var } acc)],$$

$$env_2, s \text{ with code } := c))$$

... and if exp was transformed into exp' , then

$$exp' \equiv exp + acc$$

- 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`

`...`

\Rightarrow

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

Order of evaluation

<code>evaluate (f xs ++ ys)</code>	<code>evaluate (f' xs ys)</code>
<code>xs</code>	<code>xs</code>
Body of <code>f</code>	<code>ys</code>
<code>ys</code>	Body of <code>f'</code> (‘same’ as <code>f</code>)
<code>do_app ...</code>	<code>...</code>

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`

`...`

\Rightarrow

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

Challenge:

What if `evaluate (ys) = Fail`?

`evaluate (f xs ++ ys) \neq Fail`

...

\Rightarrow

`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
```

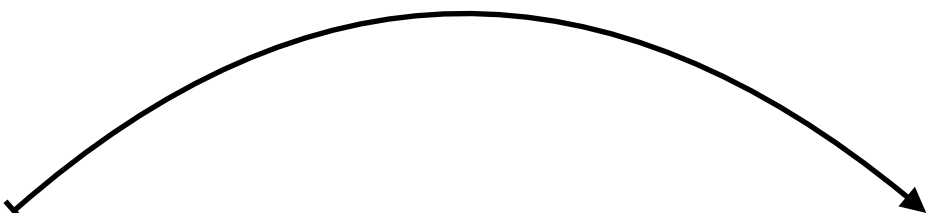
Consider the following program:

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

Equivalent BVI expression:

Raise Foo + x

What if...

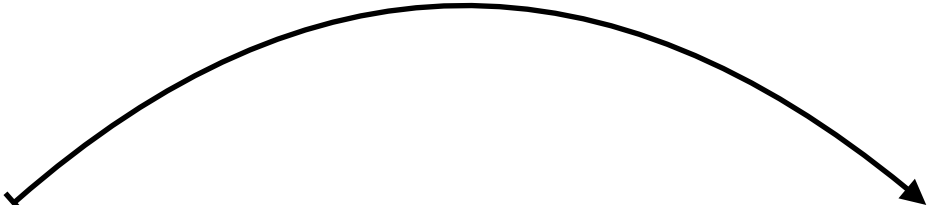


Raise Foo + x x + Raise Foo

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`

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

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

We can deduce nothing about x !

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

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

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

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

Raise Foo + x is still 'valid'

Assumption

$$(\Gamma, s) \vdash x \downarrow \text{Rerr Fail}$$

$$(\Gamma, s) \vdash x \downarrow \text{Rerr Fail}$$
$$\frac{(\Gamma, s) \vdash \text{Raise Foo} \downarrow \text{Rerr Foo} \quad (\Gamma, s) \vdash x \downarrow \text{Rerr Fail}}{(\Gamma, s) \vdash \text{Raise Foo} + x \downarrow \text{Rerr Foo}}$$
$$\frac{(\Gamma, s) \vdash x \downarrow \text{Rerr Fail} \quad (\Gamma, s) \vdash \text{Raise Foo} \downarrow \text{Rerr Foo}}{(\Gamma, s) \vdash x + \text{Raise Foo} \downarrow \text{Rerr Fail}}$$

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

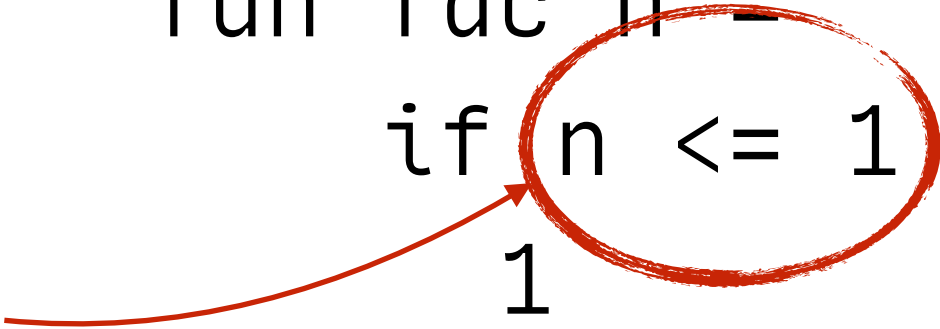
Providing guarantees

```
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*



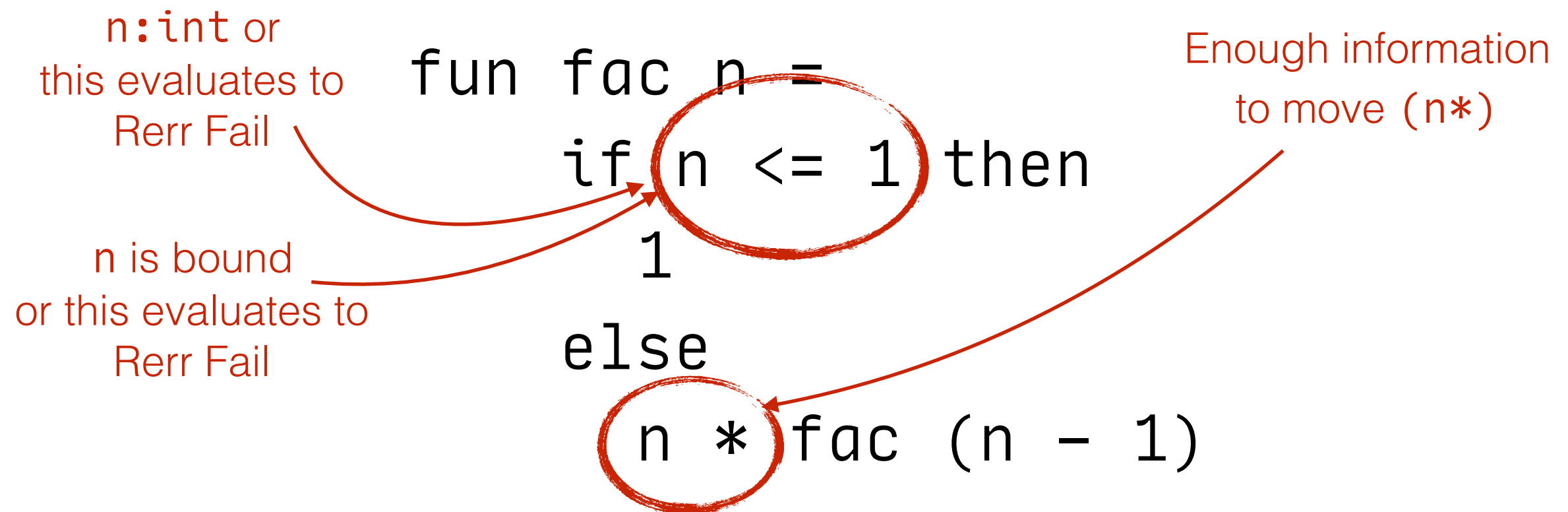
Providing guarantees

`n:int` or
this evaluates to
Rerr Fail

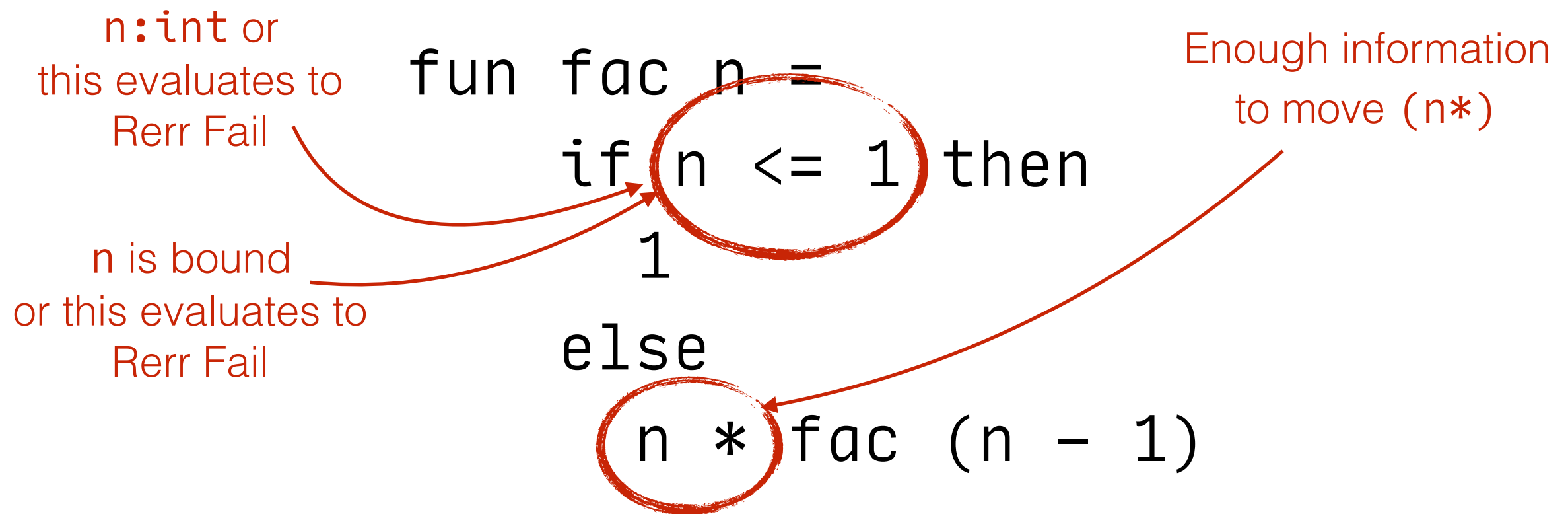
`n` is bound
or this evaluates to
Rerr Fail

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

Providing guarantees



Providing guarantees



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?