# Verification of an ML compiler

Lecture 3:
## Closures, closure conversion and call optimisations

Marktoberdorf Summer School MOD 2017

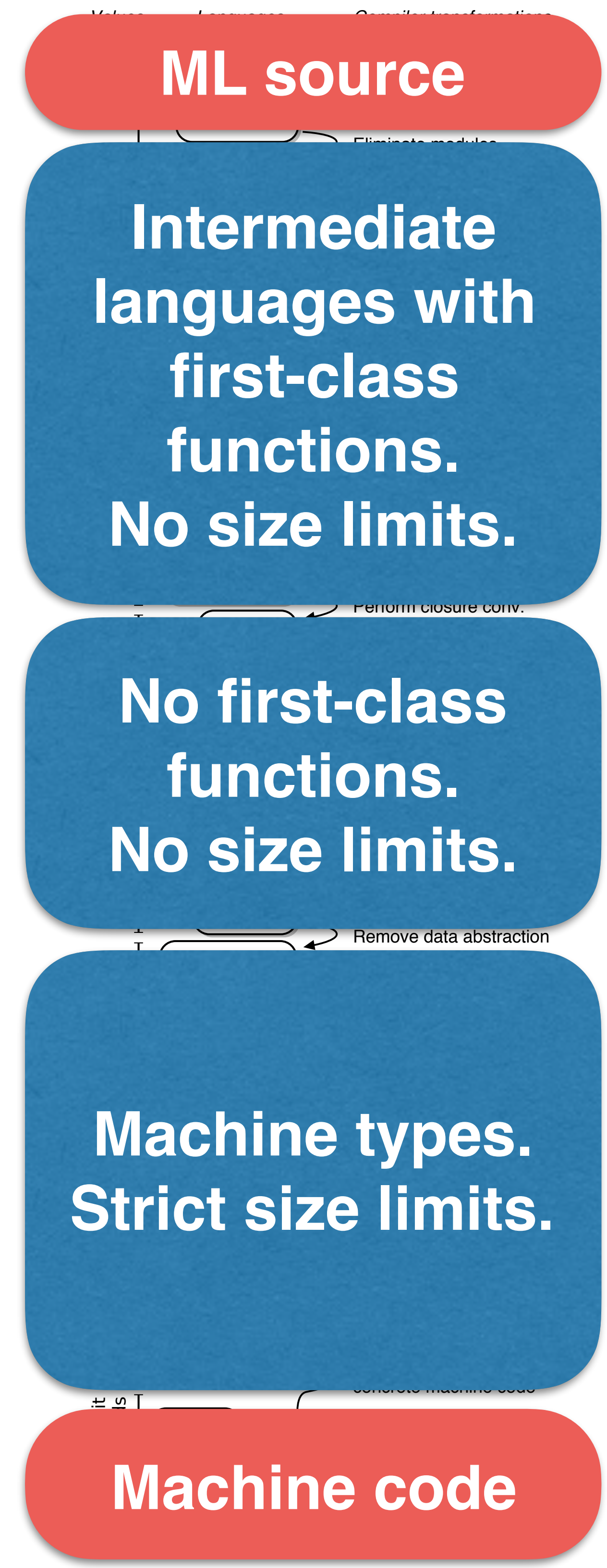
Magnus O. Myreen, Chalmers University of Technology

# Implementing the ML abstractions

The two most interesting transitions

function values are implemented (topic of this lecture)

data abstraction is implemented (topic of previous lecture)



*Value type before:*

```
v =
  Number int
| Word64 word64
| Block num (v list)
| RefPtr num
| Closure (v list) exp
| Recclosure (v list) (exp list) num
```

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

values contain code

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

**No first-class functions. No size limits.**

*Value types after:*

```
v =
  Number int
| Word64 word64
| Block num (v list)
| RefPtr num
| CodePtr num
```

# DeBruijn indexing

an index into the environment

element-of operation

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


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

value of new bound variables are prefixed to the current environment

# Semantics of closures

*Closure creation in the concrete syntax:*

```
fn v => e
```

*Evaluation in the semantics:*

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

no variable name given, since we are using dB indexing

the created closure captures the current env

# Semantics of closures (cont.)

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

*Evaluation in the semantics:*

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

evaluate function and argument

extract the closure's exp and env

evaluate the exp

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

# This lecture

Function values (called closures) bring challenges:

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

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

# This lecture

Function values (called closures) bring challenges:

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

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

# Closures cause complications

*Constant folding phase:*

```
compile (Add (Lit i) (Lit j)) = Lit (i+j)
compile (Fn e) = Fn (compile e)
...
```

*Evaluation in the semantics:*

```
  evaluate ([Add (Lit 3) (Lit 5)],env,s)
= (Rval (Number 8),s)

  evaluate ([compile (Add (Lit 3) (Lit 5))],env,s)
= evaluate ([Lit 8],env,s)
= (Rval (Number 8),s)
```

optimised code produced
the same result — ***good!***

# Closures cause complications

*Constant folding phase:*

```
compile (Add (Lit i) (Lit j)) = Lit (i+j)
compile (Fn e) = Fn (compile e)
...
```

*Evaluation in the semantics:*

```
  evaluate ([Fn (Add (Lit 3) (Lit 5))],env,s)
= ???

  evaluate ([compile (Fn (Add (Lit 3) (Lit 5)))],env,s)
= evaluate ([Fn (Lit 8)],env,s)
= ???
```

# Closures cause complications

*Constant folding phase:*

```
compile (Add (Lit i) (Lit j)) = Lit (i+j)
compile (Fn e) = Fn (compile e)
...
```

*Evaluation in the semantics:*

```
  evaluate ([Fn (Add (Lit 3) (Lit 5))],env,s)
= (Rval (Closure env (Add (Lit 3) (Lit 5))),s)

  evaluate ([compile (Fn (Add (Lit 3) (Lit 5)))],env,s)
= evaluate ([Fn (Lit 8)],env,s)
= (Rval (Closure env (Lit 8)),s)
```

Values can no longer be compared with equality

# Value relation options

*How do we relate values in presence of closures?*

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

Closure env (Lit 8)
```

code in closure must be produced by the current compiler function

*Syntactic option:*

```
 val_rel_list env1 env2        e2 = compile e1
─────────────────────────────────────────────────
  val_rel (Closure env1 e1) (Closure env2 e2)
```

*Semantic option:*

*jargon:* type-directed, step indexed, …

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

*Definition:*

```
val_rel x y    val_rel_list xs ys
----------------------------------          --------------------
val_rel_list (x::xs) (y::ys)                val_rel_list [] []
```

code in closure must be produced by the current compiler function

*Syntactic option:*

```
val_rel_list env1 env2        e2 = compile e1
----------------------------------------------
val_rel (Closure env1 e1) (Closure env2 e2)
```

*Semantic option:*

*jargon:* type-directed, step indexed, …

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

# Pros and Cons

code in closure must be produced by the current compiler function

*Syntactic option:*

```
val_rel_list env1 env2        e2 = compile e1
------------------------------------------------
val_rel (Closure env1 e1) (Closure env2 e2)
```

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

*Semantic option:*

*jargon:* type-directed, step indexed, …

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

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

# This lecture

Function values (called closures) bring challenges:

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

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

# Closure conversion

*Value type before:*

```
v =
   Number int
 | Word64 word64
 | Block num (v list)
 | RefPtr num
 | Closure (v list) exp
 | Recclosure (v list) (exp list) num
```

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

---

No first-class functions. No size limits.

*Value types after:*

```
v =
   Number int
 | Word64 word64
 | Block num (v list)
 | RefPtr num
 | CodePtr num
```

# Closure conversion

*Value type before:*

```
v =
   Number int
 | Word64 word64
 | Block num (v list)
 | RefPtr num
 | Closure (v list) exp
 | Recclosure (v list) (exp list) num
```

---

*Value types after:*

```
v =
   Number int
 | Word64 word64
 | Block num (v list)
 | RefPtr num
 | CodePtr num
```

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

# Value relation

environment list must related to values stored in Block

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

```
val_rel_list code env vals    lookup p code = compile body
---------------------------------------------------------------
   val_rel code (Closure env body)
                (Block clos_tag ([CodePtr p] ++ vals))
```

references or internal pointers are used

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

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

# Minimal environments?

env and vals are lists of same length

```
val_rel_list code env vals    lookup p code = compile body
─────────────────────────────────────────────────────────────
   val_rel code (Closure env body)
                (Block clos_tag ([CodePtr p] ++ vals))
```

Are we wasting space?

*Reminder:*

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

*Yes!* The env can contain values that are never used in e.

# Minimal environments?

env and vals are lists of same length

```
val_rel_list code env vals    lookup p code = compile body
──────────────────────────────────────────────────────────
   val_rel code (Closure env body)
                (Block clos_tag ([CodePtr p] ++ vals))
```

Are we wasting space?

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

CakeML implements this as a compiler phase right before closure conversion

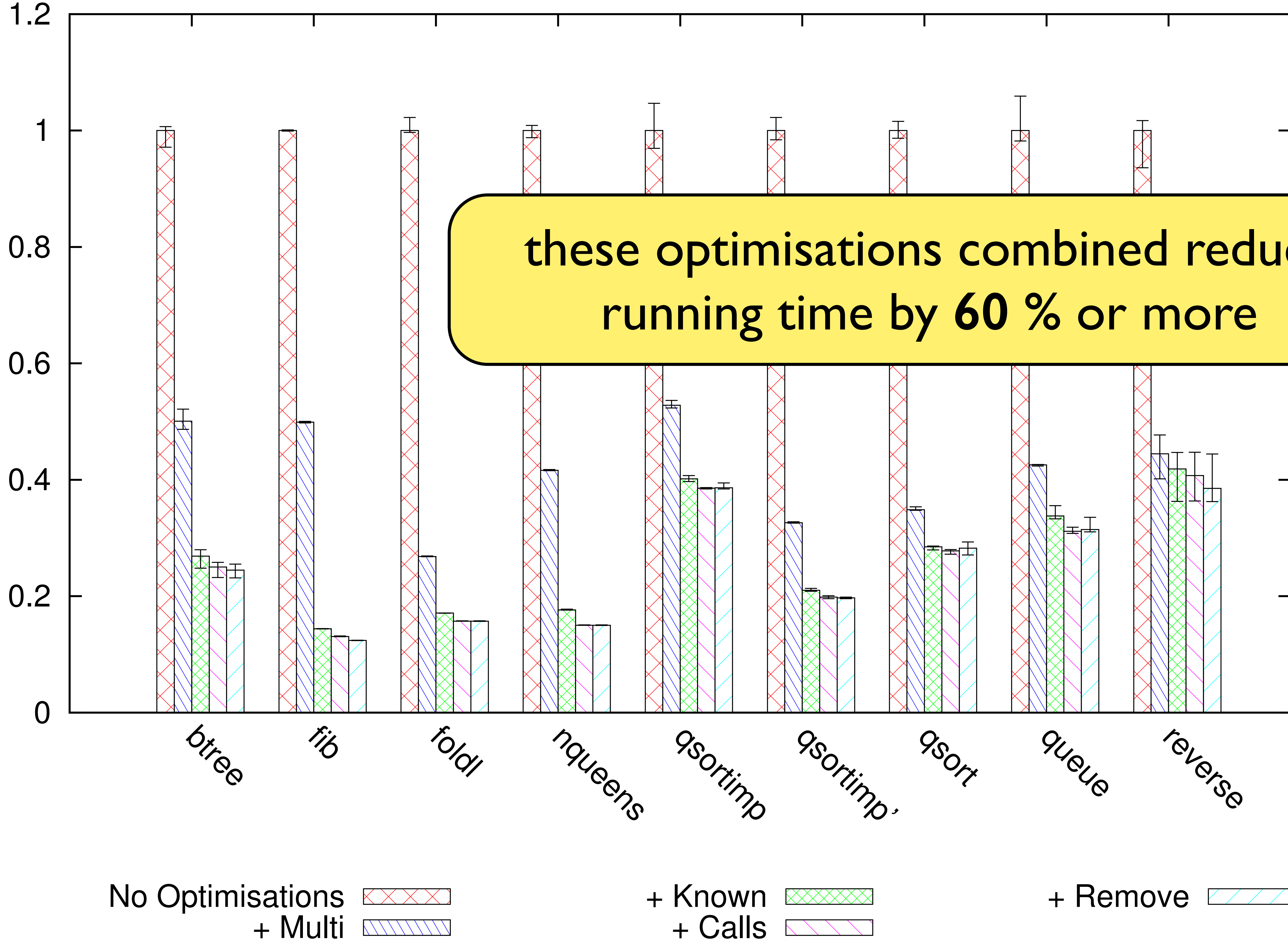
# This lecture

Function values (called closures) bring challenges:

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

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

# Optimisations with high impact

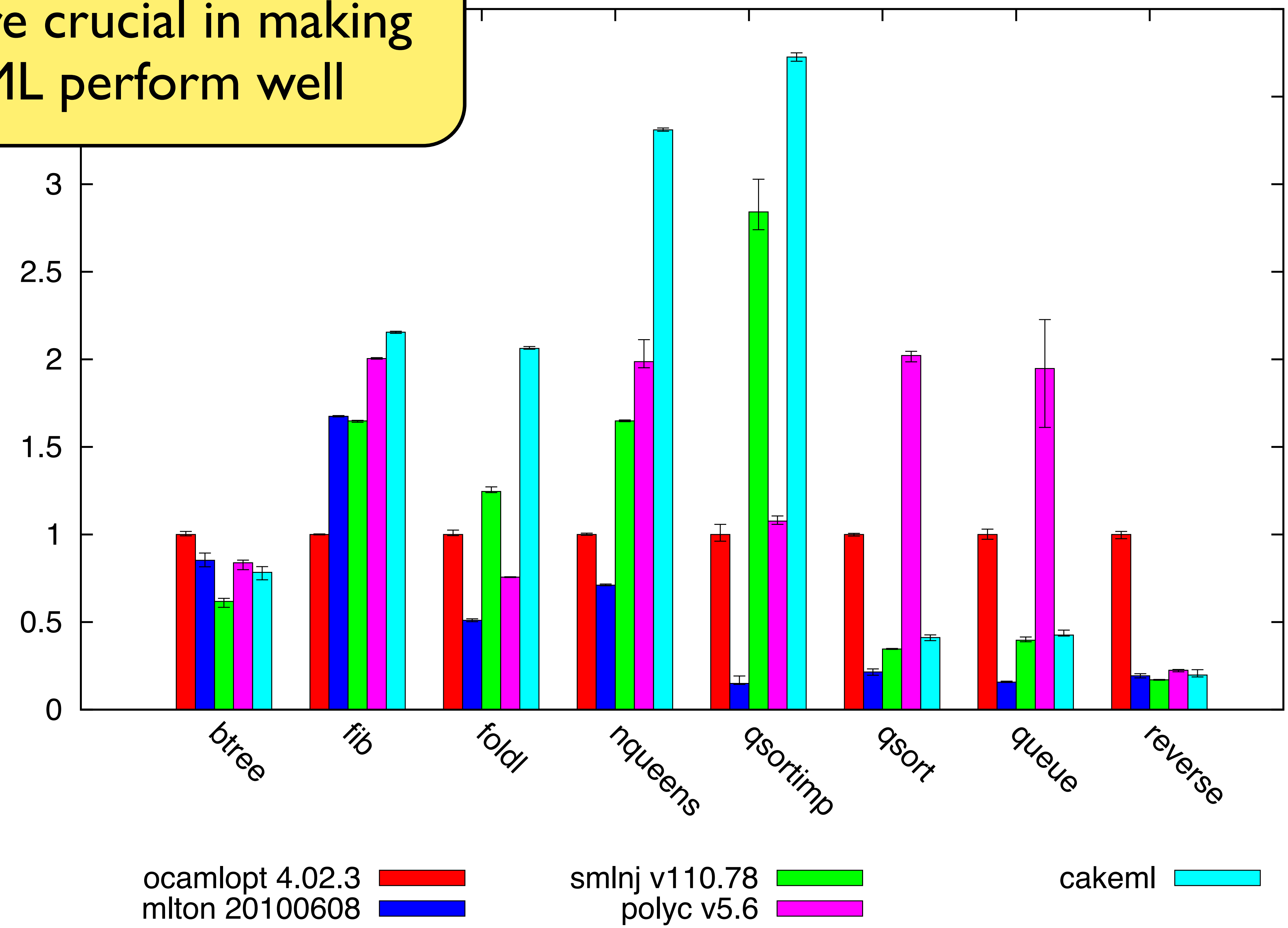



these optimisations combined reduce running time by **60 %** or more

# Comparing ML compilers

… and are crucial in making CakeML perform well

execution time relative to native-code compiled OCaml (red)



# What do the optimisations do?

***Answer:*** improve compilation of closures and calls

in fact, they try to avoid closures if possible

# Naive implementations are slow

*Example:*

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

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

*The above is syntactic sugar for:*

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

each application only consumes one argument at a time

… between each application a new closure is created

# Optimisation of function calls

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

# Optimisation of function calls

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

# Optimisation of function calls

true multi-argument closure

```
set_global 0 (fn_4 xs => let
      fun append_0 ⟨xs,ys⟩ =
        if xs = [] then ys else
          el 0 xs :: append^0 ⟨el 1 xs, ys⟩
      fun rev_2 xs =
        if xs = [] then xs else
          append^0 ⟨rev^2 (el 1 xs), [el 0 xs]⟩
  in rev^2 xs end);
set_global 1 ((get_global 0)^4 [1,2,3]);
```

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

# Optimisation of function calls

```
set_global 0 (fn xs => Call 5 ⟨xs⟩);
set_global 1 (Call 5 [1,2,3]);
```

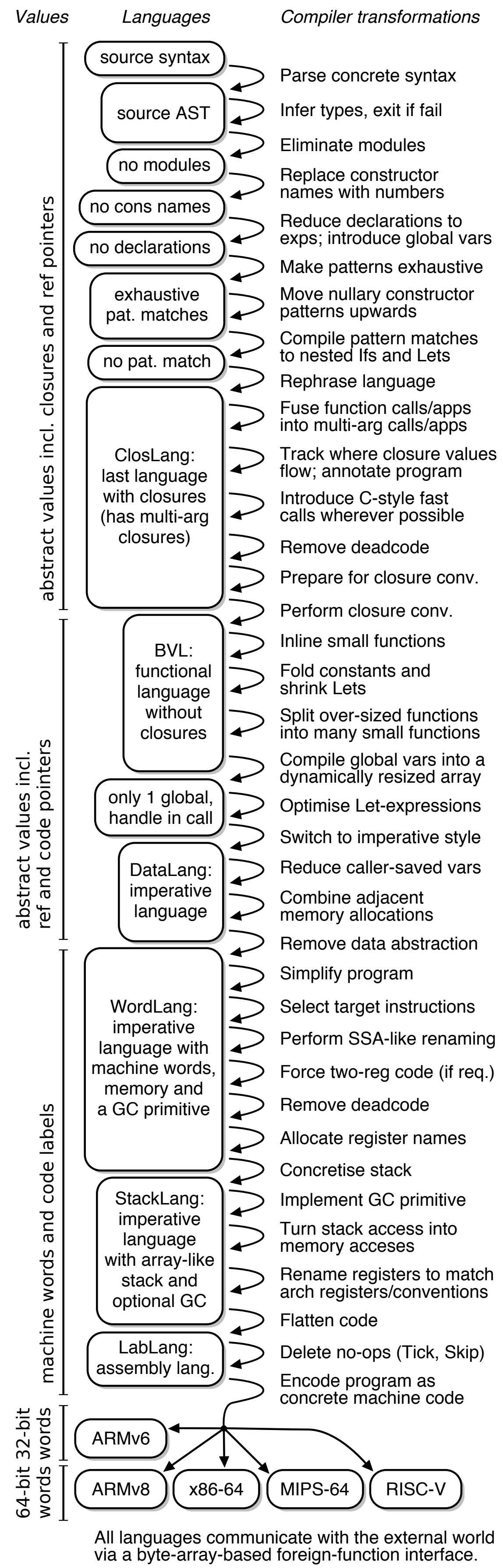
Code Table:

```
1 ⟨xs,ys⟩ => if xs = [] then ys else
                 el 0 xs :: Call 1 ⟨el 1 xs, ys⟩

3 ⟨xs⟩ => if xs = [] then xs else
             Call 1 ⟨Call 3 (el 1 xs), [el 0 xs]⟩

5 ⟨xs⟩ => let
      val append = 0
      val rev = 0
   in Call 3 ⟨xs⟩ end
```

C-like function calls

# This lecture

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




*(next slide zooms in)*

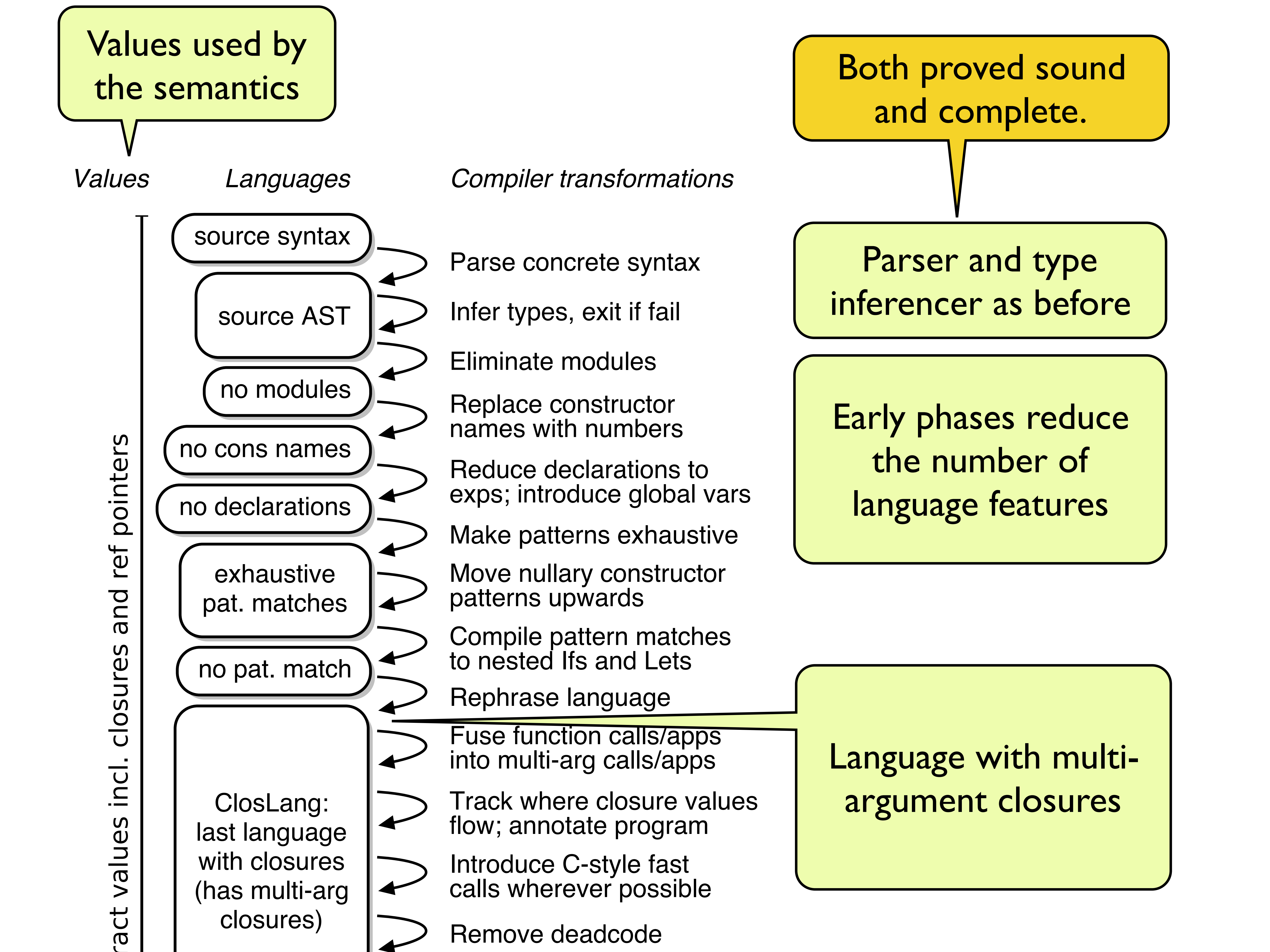
*Latest version:*

*12 intermediate languages (ILs)*

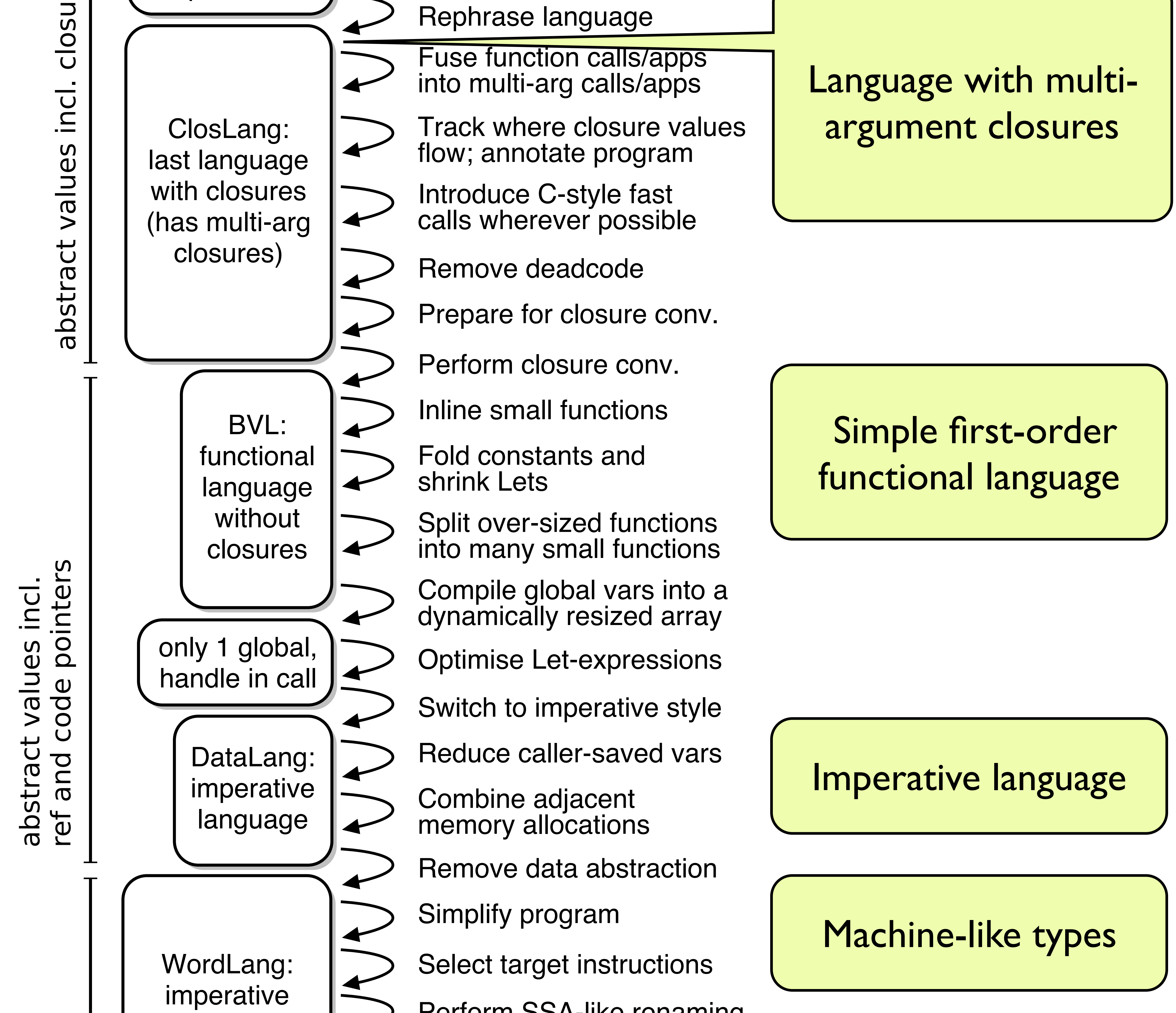
*and many within-IL optimisations*

*each IL at the right level of abstraction*

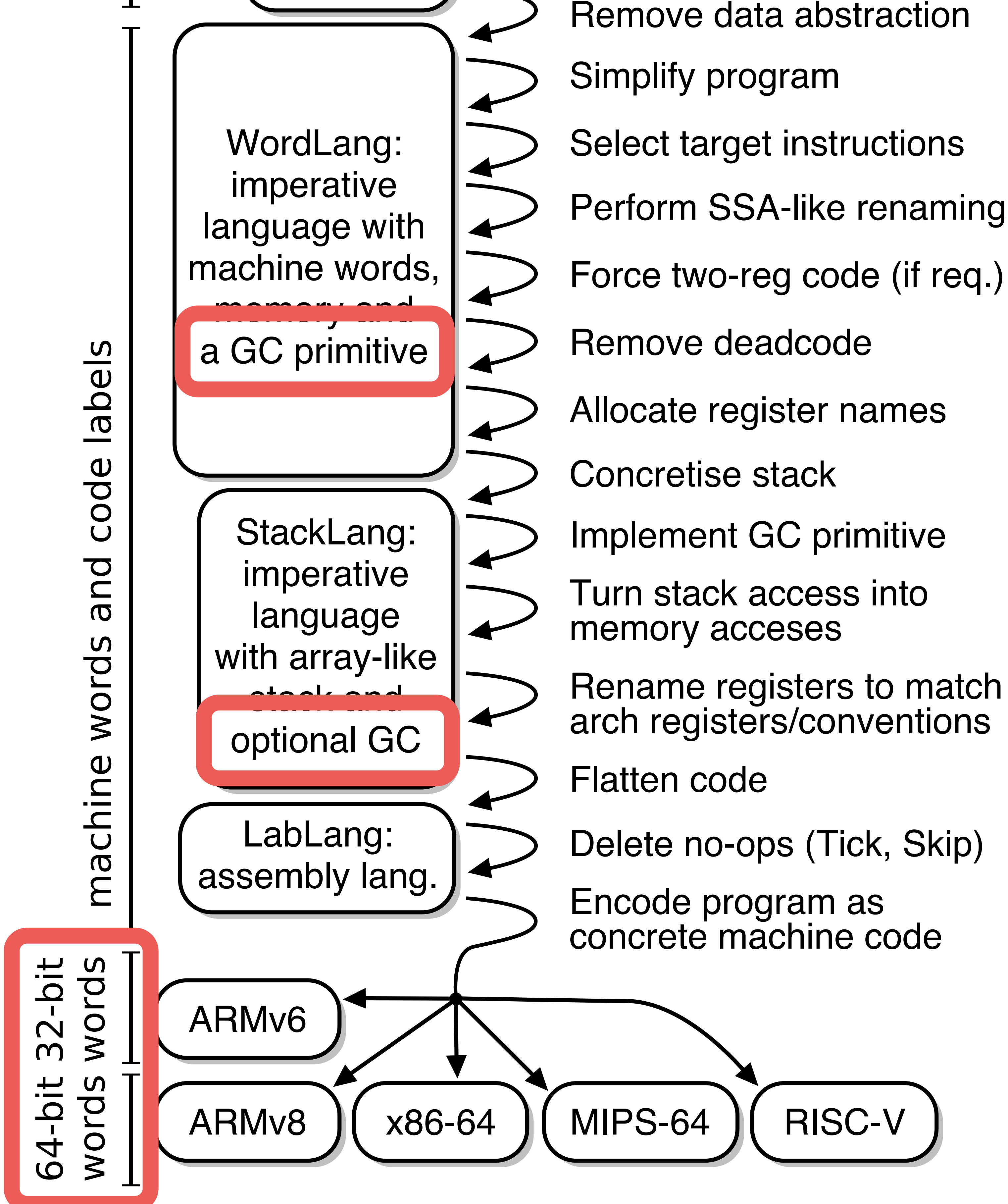
for the benefit of proofs and compiler implementation



Values used by the semantics
Both proved sound and complete.
Values
Languages
Compiler transformations
ract values incl. closures and ref pointers
source syntax
Parse concrete syntax
source AST
Infer types, exit if fail
Eliminate modules
no modules
Replace constructor names with numbers
no cons names
Reduce declarations to exps; introduce global vars
no declarations
Make patterns exhaustive
exhaustive pat. matches
Move nullary constructor patterns upwards
Compile pattern matches to nested Ifs and Lets
no pat. match
Rephrase language
Fuse function calls/apps into multi-arg calls/apps
ClosLang: last language with closures (has multi-arg closures)
Track where closure values flow; annotate program
Introduce C-style fast calls wherever possible
Remove deadcode
Parser and type inferencer as before
Early phases reduce the number of language features
Language with multi-argument closures



abstract values incl. closu
Rephrase language
ClosLang: last language with closures (has multi-arg closures)
Fuse function calls/apps into multi-arg calls/apps
Track where closure values flow; annotate program
Introduce C-style fast calls wherever possible
Remove deadcode
Prepare for closure conv.
Perform closure conv.
Language with multi-argument closures
abstract values incl. ref and code pointers
BVL: functional language without closures
Inline small functions
Fold constants and shrink Lets
Split over-sized functions into many small functions
Simple first-order functional language
Compile global vars into a dynamically resized array
only 1 global, handle in call
Optimise Let-expressions
Switch to imperative style
DataLang: imperative language
Reduce caller-saved vars
Combine adjacent memory allocations
Imperative language
Remove data abstraction
WordLang: imperative
Simplify program
Select target instructions
Machine-like types




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

Machine-like types

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

Targets 5 architectures

# What we learnt

Function values (called closures) bring challenges:

1. **Closures make stating the value relation harder** because closure values contain code, which is modified by the compiler.

   comparing values with = does not work

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

3. **Good compilation** of closures and function applications **is crucial for performance.**

# Extra slides

# Mutually recursive closures

*Value:*

```
v = ...
| Recclosure (v list) (exp list) num
```

env

list of function bodies

index: which body is to be used

*Closure creation in the concrete syntax:*

```
let fun f1 x = … and f2 = … and f3 = … in … end
```

*Evaluation in the semantics:*

```
evaluate ([Letrec funs rest],env,s) =
  evaluate ([rest], build env funs ++ env, s)
```

one binding per function

```
build env fns = Genlist (Recclosure env fns) (length fns)

Genlist f n = if 0 then [] else Genlist f (n-1) ++ [f (n-1)]
```

# Application

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

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

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

same as on earlier slide

the new part