Memory abstraction formalised, construction of verified interpreter

Lecture 6

MPhil ACS & Part III course, Functional Programming: Implementation, Specification and Verification

Magnus Myreen
Michaelmas term, 2013

Towards a verified interpreter

This lecture:

- formalise memory representation for s-expressions
- prove Hoare triples for operations over s-expressions
- revisit decompilation (use it for synthesis)
- construct an interpreter using synthesis

Towards a verified interpreter

This lecture:

- formalise memory representation for s-expressions
- prove Hoare triples for operations over s-expressions
- revisit decompilation (use it for synthesis)
- construct an interpreter using synthesis

Result: verified implementation of an FP language

Towards a verified interpreter

This lecture:

- formalise memory representation for s-expressions
- prove Hoare triples for operations over s-expressions
- revisit decompilation (use it for synthesis)
- construct an interpreter using synthesis

Result: verified implementation of an FP language

the implementation is slow
(due to interpreter-based impl.),
next lecture: *compilation!*

s-expressions revisited

Lisp's s-expressions: each s-expression is

- a number,
- a symbol, or
- a pair of s-expressions

s-expressions revisited

Lisp's s-expressions: each s-expression is

- a number, e.g. `56`
- a symbol, or e.g. `hello`
- a pair of s-expressions e.g. `(1 . 2)`

s-expressions revisited

Lisp’s s-expressions: each s-expression is

- a number, e.g. `56`
- a symbol, or e.g. `hello`
- a pair of s-expressions e.g. `(1 . 2)`

`(1 2 3)` abbreviates `(1 . (2 . (3 . nil)))`

s-expressions revisited

Lisp's s-expressions: each s-expression is

- a number, e.g. `56`
- a symbol, or e.g. `hello`
- a pair of s-expressions e.g. `(1 . 2)`

`(1 2 3)` abbreviates `(1 . (2 . (3 . nil)))`

In lecture 2, modelled as follows:

```
SExp ::= Dot SExp SExp | Val num | Sym string
```

s-expressions revisited

Lisp's s-expressions: each s-expression is

- a number, e.g. `56`
- a symbol, or e.g. `hello`
- a pair of s-expressions e.g. `(1 . 2)`

`(1 2 3)` abbreviates `(1 . (2 . (3 . nil)))`

In lecture 2, modelled as follows:

non-trivial to implement properly

```
SExp ::= Dot SExp SExp | Val num | Sym string
```

s-expressions revisited

Lisp's s-expressions: each s-expression is

- a number, e.g. `56`
- a symbol, or e.g. `hello`
- a pair of s-expressions e.g. `(1 . 2)`

`(1 2 3)` abbreviates `(1 . (2 . (3 . nil)))`

In lecture 2, modelled as follows:

non-trivial to implement properly

non-trivial to implement properly

```
SExp ::= Dot SExp SExp | Val num | Sym string
```

s-expressions revisited

Lisp's s-expressions: each s-expression is

- a number, e.g. `56`
- a symbol, or e.g. `hello`
- a pair of s-expressions e.g. `(1 . 2)`

`(1 2 3)` abbreviates `(1 . (2 . (3 . nil)))`

In lecture 2, modelled as follows:

non-trivial to implement properly

non-trivial to implement properly

```
SExp ::= Dot SExp SExp | Val num | Sym string
```

For this lecture, simplified and modelled as follows:

```
SExp ::= Dot SExp SExp | Val word30 | Sym word30
```

s-expressions revisited

Lisp's s-expressions: each s-expression is

- a number, e.g. `56`
- a symbol, or e.g. `hello`
- a pair of s-expressions e.g. `(1 . 2)`

`(1 2 3)` abbreviates `(1 . (2 . (3 . nil)))`

In lecture 2, modelled as follows:

non-trivial to implement properly

non-trivial to implement properly

```
SExp ::= Dot SExp SExp | Val num | Sym string
```

For this lecture, simplified and modelled as follows:

```
SExp ::= Dot SExp SExp | Val word30 | Sym word30
```

easy to implement

Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

A look at memory:

- memory is byte addressed (byte = 8 bits)

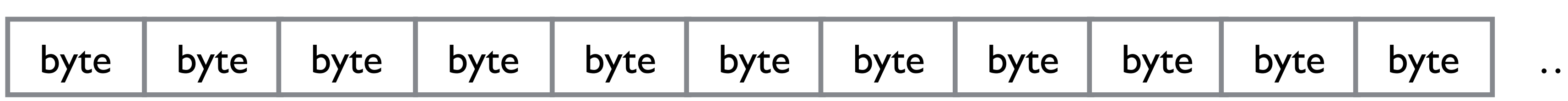
Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

A look at memory:

- memory is byte addressed (byte = 8 bits)

Sketch:



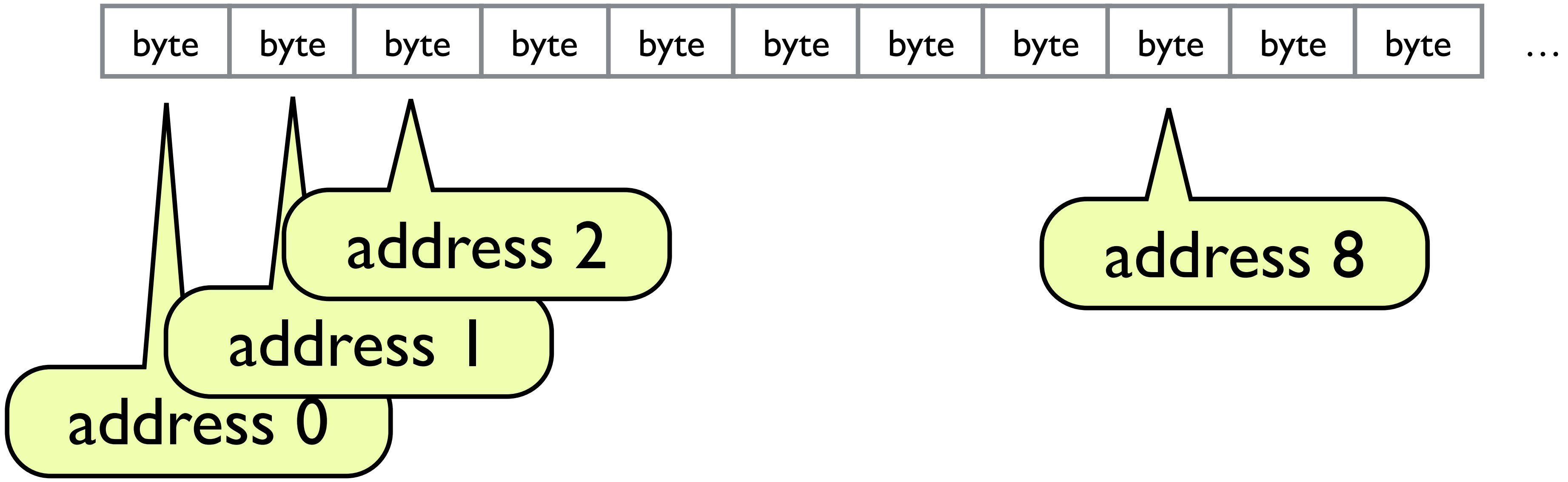
Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

A look at memory:

- memory is byte addressed (byte = 8 bits)

Sketch:



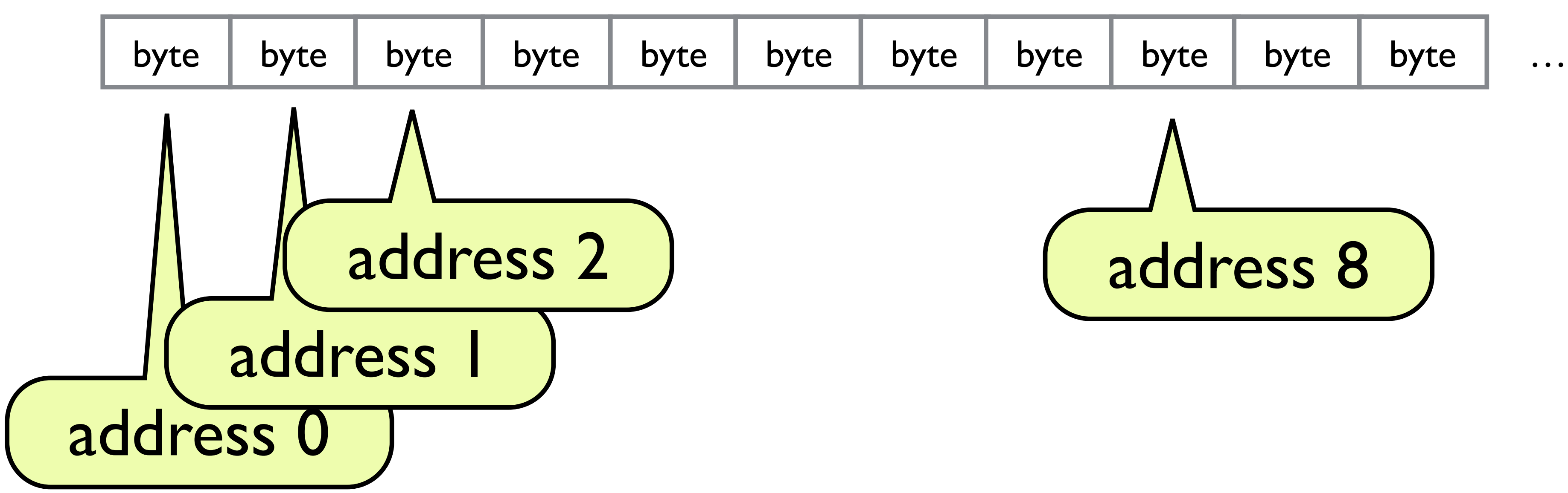
Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

A look at memory:

- memory is byte addressed (byte = 8 bits)
- 32-bit word = four bytes

Sketch:

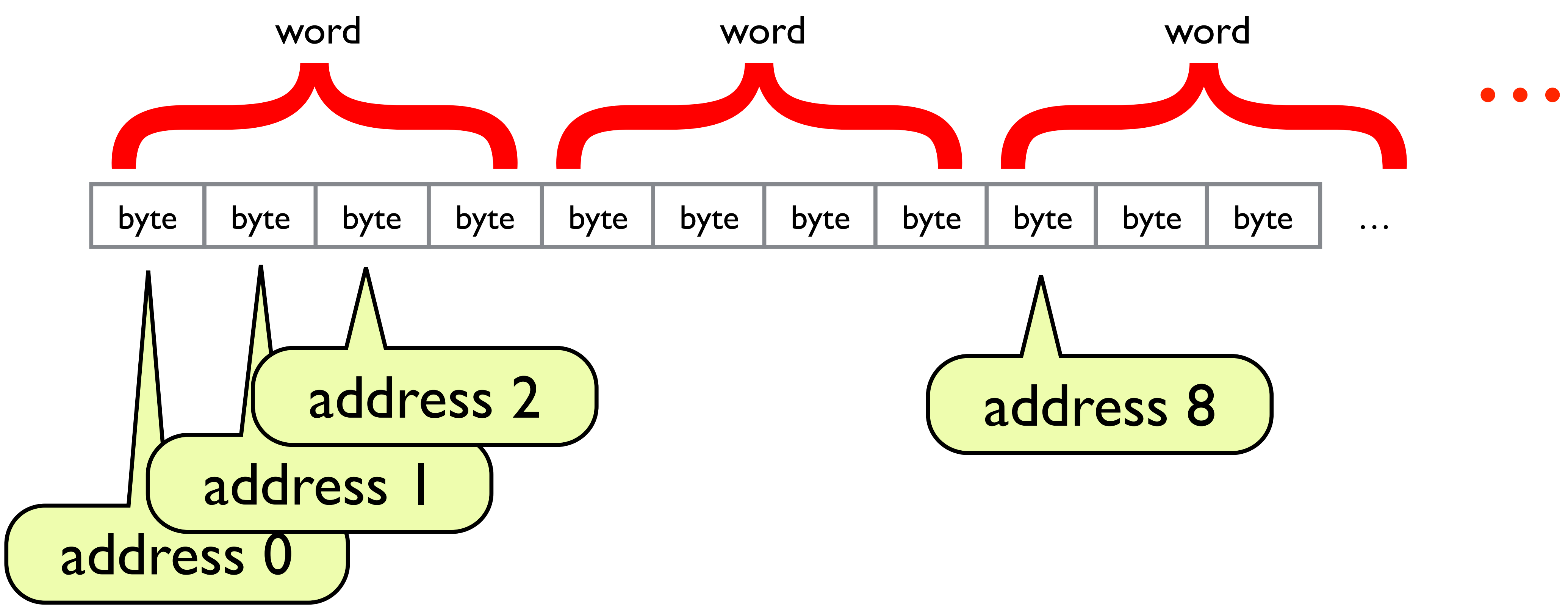


Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

A look at memory:

- memory is byte addressed (byte = 8 bits)
- 32-bit word = four bytes

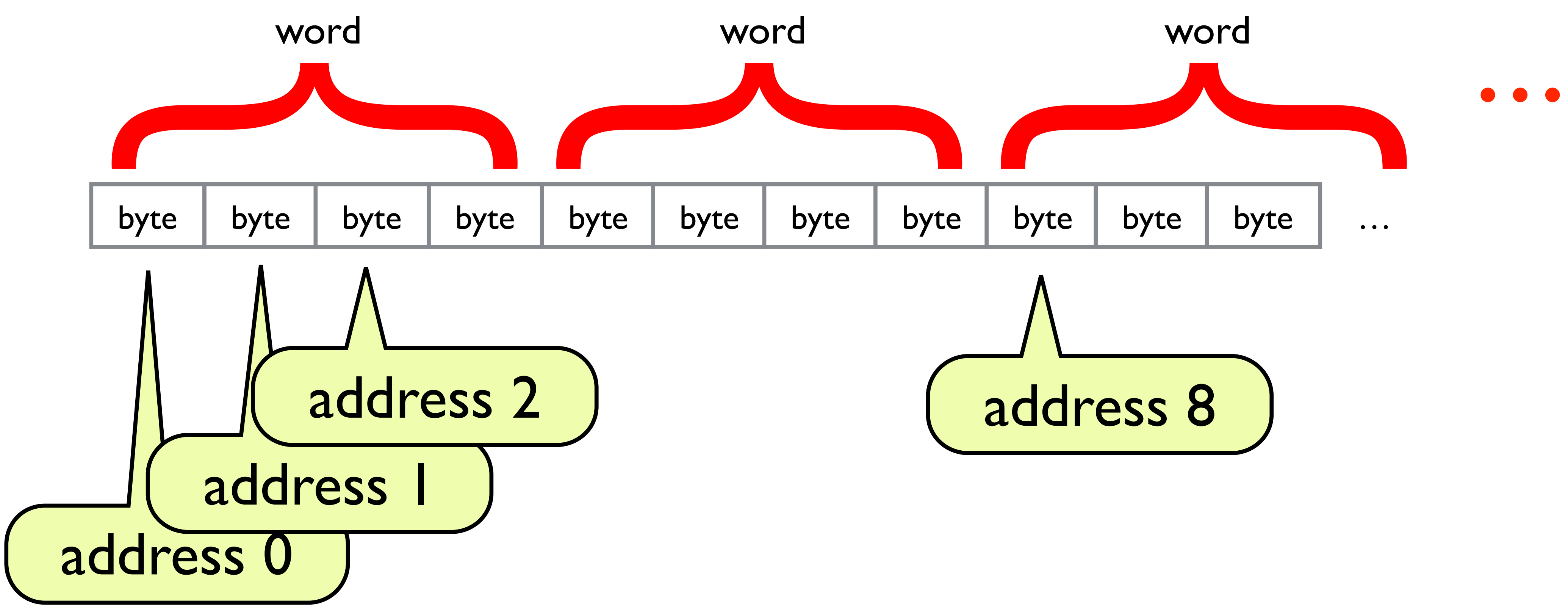


Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

A look at memory:

- memory is byte addressed (byte = 8 bits)
- 32-bit word = four bytes
- aligned words have address divisible by 4 (called 32-bit aligned)

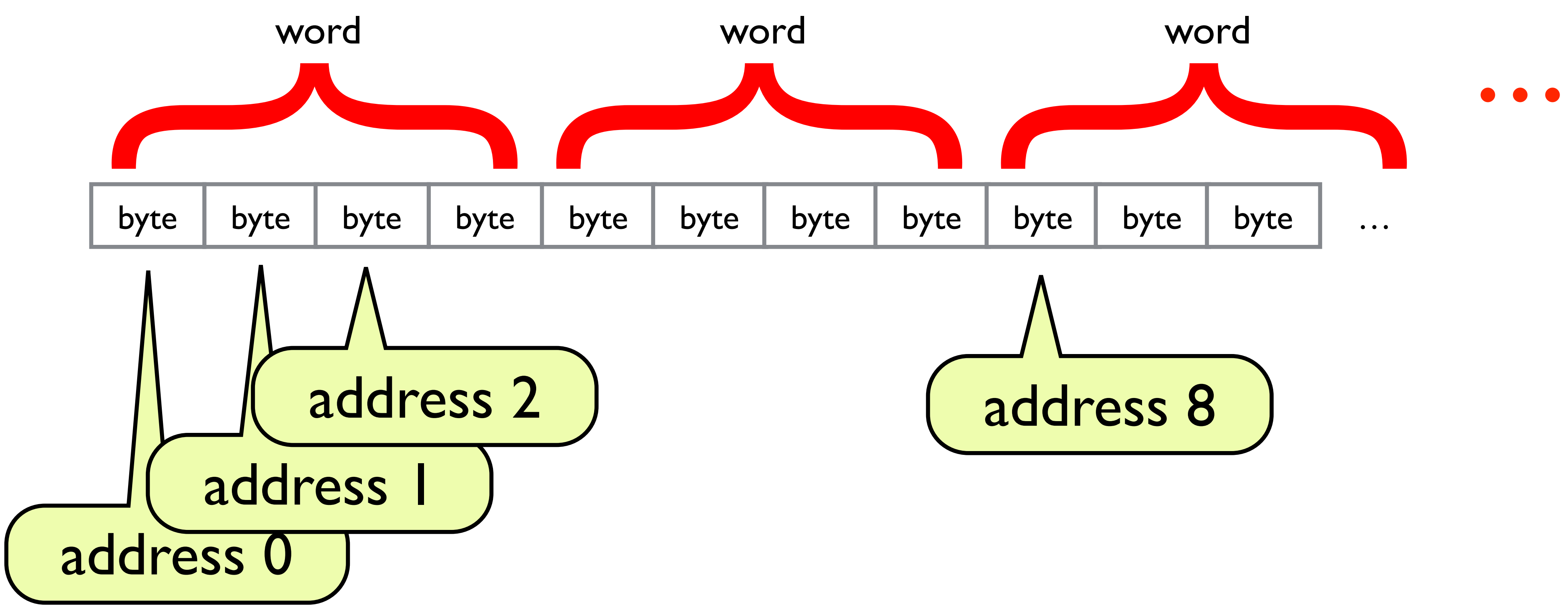


Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

A look at memory:

- memory is byte addressed (byte = 8 bits)
- 32-bit word = four bytes
- aligned words have address divisible by 4 (called 32-bit aligned)
- word-aligned pointers end in bits ‘00’



Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

Representation of s-expressions:

- cons-cells (Dot) are repr. as pointer to an aligned pair of words
- i.e. every cons-pointer ends in bits ‘00’
- numeric value v represented as word $4 \times v + 1$ (ends in bits ’01’)
- symbol value s represented as word $4 \times s + 2$ (ends in bits ’10’)

Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

Representation of s-expressions:

- cons-cells (Dot) are repr. as pointer to an aligned pair of words
- i.e. every cons-pointer ends in bits '00'
- numeric value v represented as word $4 \times v + 1$ (ends in bits '01')
- symbol value s represented as word $4 \times s + 2$ (ends in bits '10')

Example: `(1 2)` i.e. `(1 . (2 . nil))`

Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

Representation of s-expressions:

- cons-cells (Dot) are repr. as pointer to an aligned pair of words
- i.e. every cons-pointer ends in bits '00'
- numeric value v represented as word $4 \times v + 1$ (ends in bits '01')
- symbol value s represented as word $4 \times s + 2$ (ends in bits '10')

Example: `(1 2)` i.e. `(1 . (2 . nil))`

Representation: 200

Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

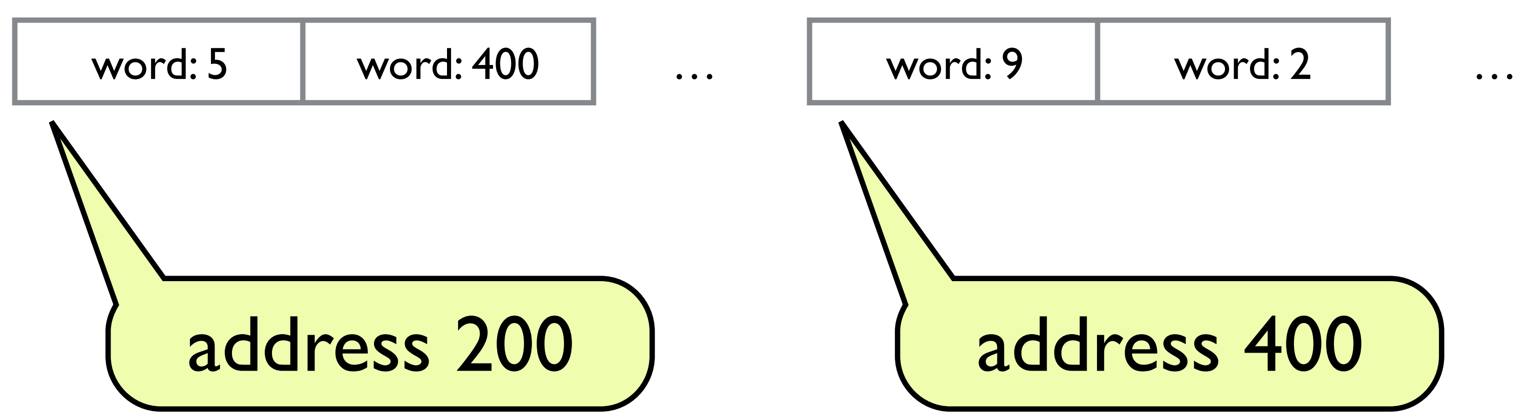
Representation of s-expressions:

- cons-cells (Dot) are repr. as pointer to an aligned pair of words
- i.e. every cons-pointer ends in bits '00'
- numeric value v represented as word $4 \times v + 1$ (ends in bits '01')
- symbol value s represented as word $4 \times s + 2$ (ends in bits '10')

Example: `(1 2)` i.e. `(1 . (2 . nil))`

Representation: 200

With memory:



Bits, Bytes, Words and Memory

Trick: lower bits used to distinguish between pointers and data.

Representation of s-expressions:

- cons-cells (Dot) are repr. as pointer to an aligned pair of words
- i.e. every cons-pointer ends in bits '00'
- numeric value v represented as word 4 × v + 1 (ends in bits '01')
- symbol value s represented as word 4 × s + 2 (ends in bits '10')

Example: `(1 2)` i.e. `(1 . (2 . nil))`

'nil' symbol number 0

Representation: 200

With memory:

word: 5	word: 400	…	word: 9	word: 2	…

address 200

address 400

Representation formalised

Formalised:

```
    sexp_repr (w,m,Val n) = (w = 4 × n + 1)

    sexp_repr (w,m,Sym s) = (w = 4 × s + 2)

sexp_repr (w,m,Dot x1 x2) = w+0 ∈ domain m ∧ sexp_repr (m(w+0),m,x1) ∧
                            w+4 ∈ domain m ∧ sexp_repr (m(w+4),m,x2) ∧
                            word_aligned w
```

Representation formalised

32-bit word

memory as partial map from word to word

Formalised:

```
    sexp_repr (w,m,Val n) = (w = 4 × n + 1)

    sexp_repr (w,m,Sym s) = (w = 4 × s + 2)

sexp_repr (w,m,Dot x1 x2) = w+0 ∈ domain m ∧ sexp_repr (m(w+0),m,x1) ∧
                            w+4 ∈ domain m ∧ sexp_repr (m(w+4),m,x2) ∧
                            word_aligned w
```

Representation formalised

32-bit word

memory as partial map from word to word

Formalised:

```
    sexp_repr (w,m,Val n) = (w = 4 × n + 1)
```

content of memory

```
    sexp_repr (w,m,Sym s) = (w = 4 × s + 2)

sexp_repr (w,m,Dot x1 x2) = w+0 ∈ domain m ∧ sexp_repr (m(w+0),m,x1) ∧
                            w+4 ∈ domain m ∧ sexp_repr (m(w+4),m,x2) ∧
                            word_aligned w
```

Representation formalised

32-bit word

memory as partial map from word to word

Formalised:

```
   sexp_repr (w,m,Val n) = (w = 4 × n + 1)

   sexp_repr (w,m,Sym s) = (w = 4 × s + 2)
```

content of memory

```
sexp_repr (w,m,Dot x1 x2) = w+0 ∈ domain m ∧ sexp_repr (m(w+0),m,x1) ∧
                            w+4 ∈ domain m ∧ sexp_repr (m(w+4),m,x2) ∧
                            word_aligned w
```

Restricted to a range {i..j} of addresses in memory:

```
sexp_range (w,m,i,j,x) = sexp_repr (w,m↾{a | i ≤ a ∧ a < j},x)
```

Representation formalised

32-bit word

memory as partial map from word to word

Formalised:

```
    sexp_repr (w,m,Val n) = (w = 4 × n + 1)

    sexp_repr (w,m,Sym s) = (w = 4 × s + 2)

sexp_repr (w,m,Dot x1 x2) = w+0 ∈ domain m ∧ sexp_repr (m(w+0),m,x1) ∧
                            w+4 ∈ domain m ∧ sexp_repr (m(w+4),m,x2) ∧
                            word_aligned w
```

content of memory

Restricted to a range {i..j} of addresses in memory:

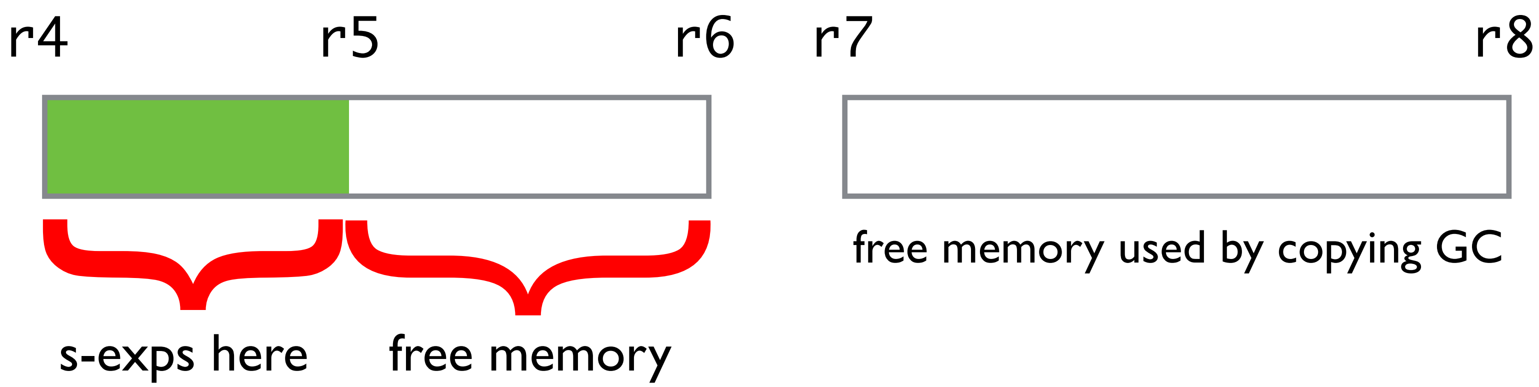
```
sexp_range (w,m,i,j,x) = sexp_repr (w,m↾{a | i ≤ a ∧ a < j},x)
```

forces s-expression x be stored in i..j range of addresses in memory

Interpreter's state

State setup:

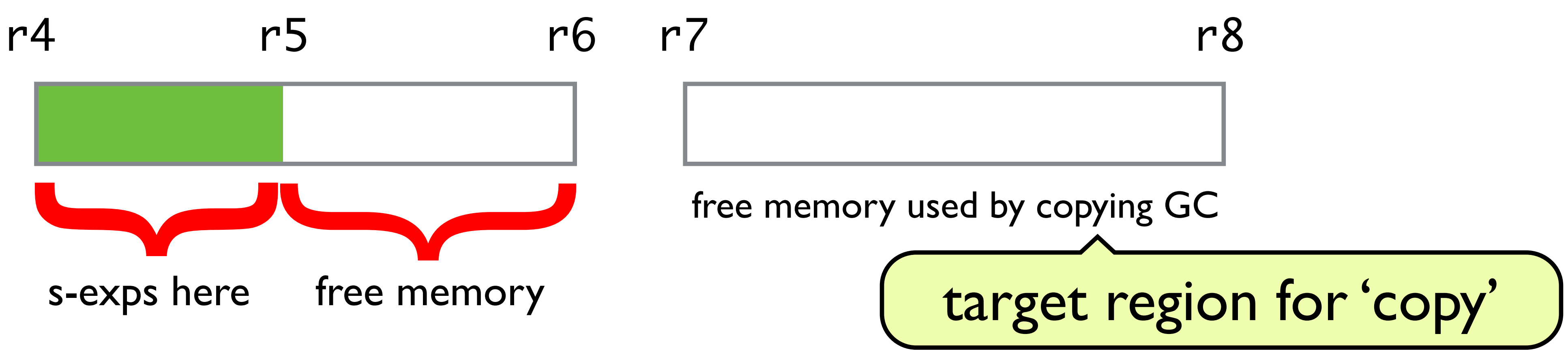
- 4 s-expressions (held in registers `r0`,`r1`,`r2`,`r3`)
- memory split into two heaps (only one active at a time)



Interpreter's state

State setup:

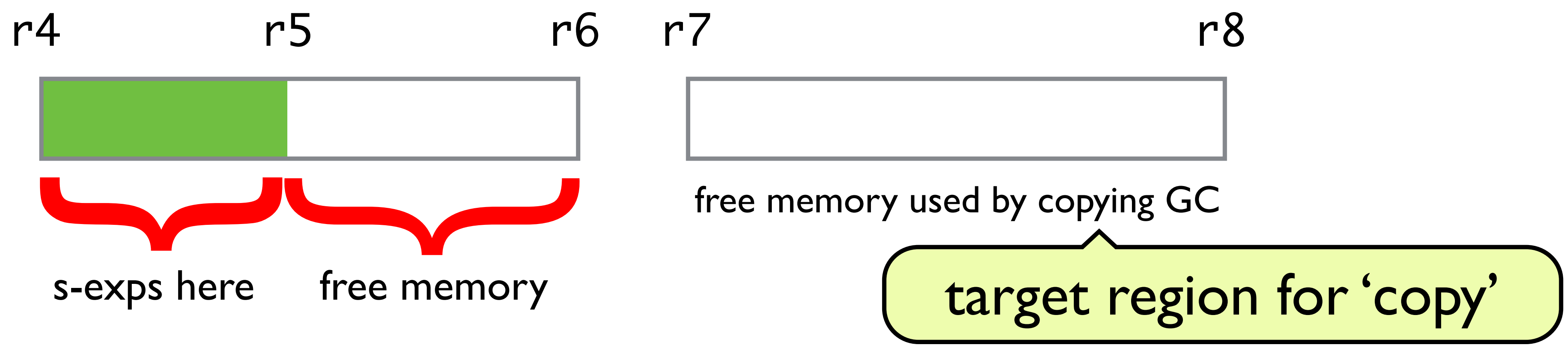
- 4 s-expressions (held in registers `r0`,`r1`,`r2`,`r3`)
- memory split into two heaps (only one active at a time)



Interpreter's state

State setup:

- 4 s-expressions (held in registers r0,r1,r2,r3)
- memory split into two heaps (only one active at a time)

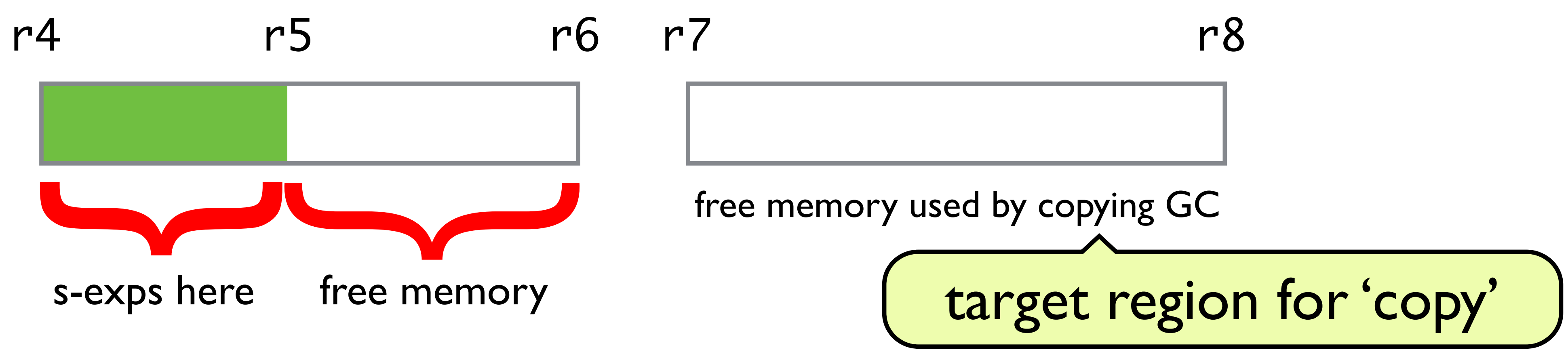



```
heap_inv (x0,x1,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8) =
  sexp_range (r0,m,r4,r5,x0) ∧ sexp_range (r1,m,r4,r5,x1) ∧
  sexp_range (r2,m,r4,r5,x2) ∧ sexp_range (r3,m,r4,r5,x3) ∧
  domain m = { a | r4 ≤ a ∧ a < r6 } ∪ { a | r7 ≤ a ∧ a < r8 } ∧
  0 ≤ r4 ≤ r5 ≤ r6 ∧ 0 ≤ r7 ≤ r8 ∧ (r8 - r7 = r6 - r4)
```

Interpreter's state

State setup:

- 4 s-expressions (held in registers r0,r1,r2,r3)
- memory split into two heaps (only one active at a time)



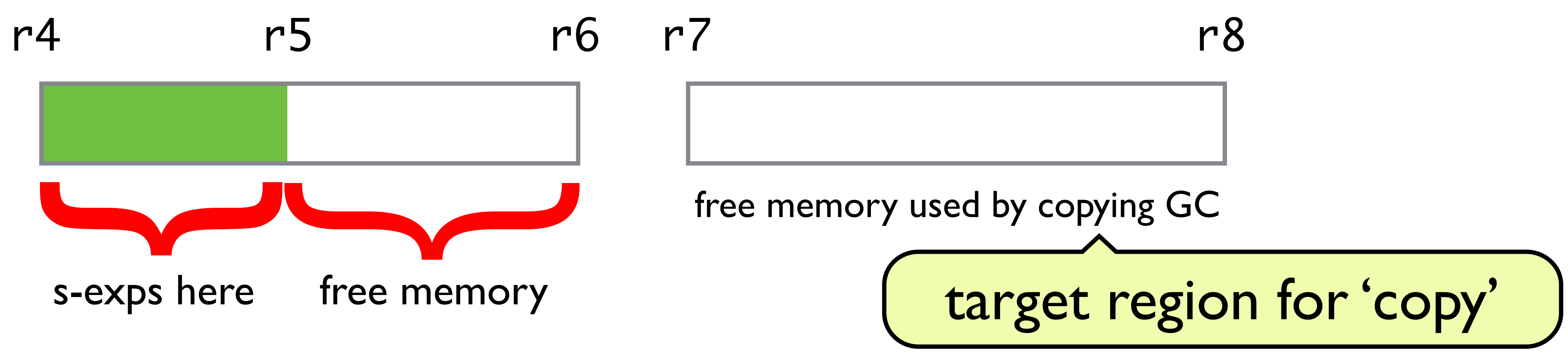

```
heap_inv (x0,x1,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8) =
  sexp_range (r0,m,r4,r5,x0) ∧ sexp_range (r1,m,r4,r5,x1) ∧
  sexp_range (r2,m,r4,r5,x2) ∧ sexp_range (r3,m,r4,r5,x3) ∧
  domain m = { a | r4 ≤ a ∧ a < r6 } ∪ { a | r7 ≤ a ∧ a < r8 } ∧
  0 ≤ r4 ≤ r5 ≤ r6 ∧    ≤ r7 ≤ r8 ∧ (r8 - r7 = r6 - r4)
```

s-exps live within addresses r4…r5

Interpreter's state

State setup:

- 4 s-expressions (held in registers r0,r1,r2,r3)
- memory split into two heaps (only one active at a time)




```
heap_inv (x0,x1,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8) =
  sexp_range (r0,m,r4,r5,x0) ∧ sexp_range (r1,m,r4,r5,x1) ∧
  sexp_range (r2,m,r4,r5,x2) ∧ sexp_range (r3,m,r4,r5,x3) ∧
  domain m = { a | r4 ≤ a ∧ a < r6 } ∪ { a | r7 ≤ a ∧ a < r8 } ∧
  0 ≤ r4 ≤ r5 ≤ r6 ∧ r6 ≤ r7 ≤ r8 ∧ (r8 - r7 = r6 - r4)
```

s-exps live within addresses r4…r5

two heaps of equal length

Heap assertion

We want to prove Hoare triples:

- define state assertion using `MEM`, `R` etc.
- and ∃ over state

Heap assertion

We want to prove Hoare triples:

- define state assertion using `MEM`, `R` etc.
- and ∃ over state

```
HEAP (x0,x1,x2,x3,err) =
  ∃m r0 r1 r2 r3 r4 r5 r6 r7 r8.
    MEM m * R 0 r0 * R 1 r1 * … * R 8 r8 * R 9 err *
    pure (heap_inv (x0,x1,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8))
```

Heap assertion

We want to prove Hoare triples:

- define state assertion using `MEM`, `R` etc.
- and ∃ over state

```
HEAP (x0,x1,x2,x3,err) =
  ∃m r0 r1 r2 r3 r4 r5 r6 r7 r8.
    MEM m * R 0 r0 * R 1 r1 * … * R 8 r8 * R 9 err *
    pure (heap_inv (x0,x1,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8))
```

∃ over state

Heap assertion

We want to prove Hoare triples:

- define state assertion using MEM, R etc.
- and ∃ over state

```
HEAP (x0,x1,x2,x3,err) =
  ∃m r0 r1 r2 r3 r4 r5 r6 r7 r8.
    MEM m * R 0 r0 * R 1 r1 * … * R 8 r8 * R 9 err *
    pure (heap_inv (x0,x1,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8))
```

∃ over state

In HOL, we can define new quantifiers:

```
(∃x. P x) = λs. ∃x. P x s
```

Heap assertion

We want to prove Hoare triples:

- define state assertion using MEM, R etc.
- and ∃ over state

```
HEAP (x0,x1,x2,x3,err) =
  ∃m r0 r1 r2 r3 r4 r5 r6 r7 r8.
    MEM m * R 0 r0 * R 1 r1 * … * R 8 r8 * R 9 err *
    pure (heap_inv (x0,x1,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8))
```

∃ over state

In HOL, we can define new quantifiers:

```
(∃x. P x) = λs. ∃x. P x s
```

or: `(∃) = λP s. ∃x. P x s`

Hoare triples with Heap assert

We can describe certain machine code w.r.t. HEAP assertion.

Hoare triples with Heap assert

We can describe certain machine code w.r.t. HEAP assertion.

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  “ mov r0,r1 ”
{ PC (pc + 4) * HEAP (x1,x1,x2,x3,err) }
```

Hoare triples with Heap assert

We can describe certain machine code w.r.t. HEAP assertion.

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  “ mov r0,r1 ”
{ PC (pc + 4) * HEAP (x1,x1,x2,x3,err) }
```

implements: `x0 := x1`

Hoare triples with Heap assert

We can describe certain machine code w.r.t. HEAP assertion.

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  “ mov r0,r1 ”
{ PC (pc + 4) * HEAP (x1,x1,x2,x3,err) }
```

implements: `x0 := x1`

```
(∃x y. x1 = Dot x y)  ⟹

{ PC pc * HEAP (x0,x1,x2,x3,err) }
  “ load r2,[r1] ”
{ PC (pc + 4) * HEAP (x0,x1,car x1,x3,err) }

where car (Dot x y) = x
```

Hoare triples with Heap assert

We can describe certain machine code w.r.t. HEAP assertion.

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  “ mov r0,r1 ”
{ PC (pc + 4) * HEAP (x1,x1,x2,x3,err) }
```

implements: x0 := x1

```
(∃x y. x1 = Dot x y)  ⟹

{ PC pc * HEAP (x0,x1,x2,x3,err) }
  “ load r2,[r1] ”
{ PC (pc + 4) * HEAP (x0,x1,car x1,x3,err) }
```

implements: x2 := car x1

```
where car (Dot x y) = x
```

Hoare triples with Heap assert

We can describe certain machine code w.r.t. HEAP assertion.

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  “ mov r0,r1 ”
{ PC (pc + 4) * HEAP (x1,x1,x2,x3,err) }
```

implements: `x0 := x1`

```
(∃x y. x1 = Dot x y)  ⟹

{ PC pc * HEAP (x0,x1,x2,x3,err) }
  “ load r2,[r1] ”
{ PC (pc + 4) * HEAP (x0,x1,car x1,x3,err) }

where car (Dot x y) = x
```

implements: `x2 := car x1`

HEAP assertion makes machine seem to operate over 4 s-exprs.

Proof of Hoare triple

Let's prove:

```
(∃x y. x1 = Dot x y)  ⟹

{ PC pc * HEAP (x0,x1,x2,x3,err) }
  “ load r2,[r1] ”
{ PC (pc + 4) * HEAP (x0,x1,car x1,x3,err) }

where car (Dot x y) = x
```

Proof of Hoare triple

Let's prove:

```
(∃x y. x1 = Dot x y)  ⟹

{ PC pc * HEAP (x0,x1,x2,x3,err) }
  “ load r2,[r1] ”
{ PC (pc + 4) * HEAP (x0,x1,car x1,x3,err) }

where car (Dot x y) = x
```

Equivalent to:

```
{ PC pc * HEAP (x0,Dot x y,x2,x3,err) }
  “ load r2,[r1] ”
{ PC (pc + 4) * HEAP (x0,Dot x y,x,x3,err) }
```

Proof of Hoare triple (cont.)

Want to show:

```
{ PC pc * HEAP (x0,Dot x y,x2,x3,err) }
  “ load r2,[r1] ”
{ PC (pc + 4) * HEAP (x0,Dot x y,x,x3,err) }
```

Proof of Hoare triple (cont.)

Want to show:

```
{ PC pc * HEAP (x0,Dot x y,x2,x3,err) }
  " load r2,[r1] "
{ PC (pc + 4) * HEAP (x0,Dot x y,x,x3,err) }
```

We start from basic Hoare triple:

```
{ PC pc * R 1 r1 * R 2 r2 * MEM m * pure (r1 ∈ domain m) }
  " load r2,[r1] "
{ PC (pc + 4) * R 1 r1 * R 2 (m(r1)) * MEM m }
```

Proof of Hoare triple (cont.)

Want to show:

```
{ PC pc * HEAP (x0,Dot x y,x2,x3,err) }
  " load r2,[r1] "
{ PC (pc + 4) * HEAP (x0,Dot x y,x,x3,err) }
```

We start from basic Hoare triple:

```
{ PC pc * R 1 r1 * R 2 r2 * MEM m * pure (r1 ∈ domain m) }
  " load r2,[r1] "
{ PC (pc + 4) * R 1 r1 * R 2 (m(r1)) * MEM m }
```

Proof plan:

1. use Frame rule to extend Hoare triple, then
2. weaken postcondition to match,
3. introduce ∃ in precondition, and finally
4. strengthen precondition.

Details of proof

We start from basic Hoare triple:

```
{ PC pc * R 1 r1 * R 2 r2 * MEM m * pure (r1 ∈ domain m) }
  " load r2,[r1] "
{ PC (pc + 4) * R 1 r1 * R 2 (m(r1)) * MEM m }
```

After application of Frame rule:

```
{ PC pc * R 1 r1 * R 2 r2 * MEM m * pure (r1 ∈ domain m) *
  R 0 r0 * R 3 r3 * … * R 8 r8 * R 9 err *
  pure (heap_inv (x0,Dot x y,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8)) }
  " load r2,[r1] "
{ PC (pc + 4) * R 1 r1 * R 2 (m(r1)) * MEM m *
  R 0 r0 * R 3 r3 * … * R 8 r8 * R 9 err *
  pure (heap_inv (x0,Dot x y,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8)) }
```

Details of proof (cont.)

We apply postcondition weakening i.e.

```
{ P } C { Q } ∧ (∀s. Q s ⇒ R s) ⇒ { P } C { R }
```

to prove:

```
{ PC pc * R 1 r1 * R 2 r2 * MEM m * pure (r1 ∊ domain m) *
  R 0 r0 * R 3 r3 * … * R 8 r8 * R 9 err *
  pure (heap_inv (x0,Dot x y,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8)) }
  “ load r2,[r1] ”
{ PC (pc + 4) * HEAP (x0,Dot x y,x,x3,err) }
```

This step required proving:

```
∀s. (PC (pc + 4) * R 1 r1 * R 2 (m(r1)) * MEM m *
     R 0 r0 * R 3 r3 * … * R 8 r8 * R 9 err *
     pure (heap_inv (x0,Dot x y,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8))) s ⇒
    (PC (pc + 4) * HEAP (x0,Dot x y,x,x3,err)) s
```

Details of proof (cont.)

Next, introduce ∃ in precondition using

```
(∀x. { P x } C { Q }) ⇔ { ∃x. P x } C { R }
```

to prove:

```
{ ∃m r0 r1 r2 r3 r4 r5 r6 r7 r8.
    PC pc * R 1 r1 * R 2 r2 * MEM m * pure (r1 ∈ domain m) *
    R 0 r0 * R 3 r3 * … * R 8 r8 * R 9 err *
    pure (heap_inv (x0,Dot x y,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8)) }
  “ load r2,[r1] ”
{ PC (pc + 4) * HEAP (x0,Dot x y,x,x3,err) }
```

Details of proof (cont.)

Next, introduce ∃ in precondition using

```
(∀x. { P x } C { Q }) ⇔ { ∃x. P x } C { R }
```

x only appears in P, not in C or Q

to prove:

```
{ ∃m r0 r1 r2 r3 r4 r5 r6 r7 r8.
    PC pc * R 1 r1 * R 2 r2 * MEM m * pure (r1 ∈ domain m) *
    R 0 r0 * R 3 r3 * … * R 8 r8 * R 9 err *
    pure (heap_inv (x0,Dot x y,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8)) }
  “ load r2,[r1] ”
{ PC (pc + 4) * HEAP (x0,Dot x y,x,x3,err) }
```

Details of proof (cont.)

We apply precondition strengthening i.e.

```
{ P } C { Q } ∧ (∀s. R s ⇒ P s) ⇒ { R } C { Q }
```

to prove:

```
{ PC pc * HEAP (x0,Dot x y,x2,x3,err) }
  “ load r2,[r1] ”
{ PC (pc + 4) * HEAP (x0,Dot x y,x,x3,err) }
```

This step required proving:

```
∀s. (PC pc * HEAP (x0,Dot x y,x2,x3,err)) s ⇒
    (∃m r0 r1 r2 r3 r4 r5 r6 r7 r8.
      PC pc * R 1 r1 * R 2 r2 * MEM m * pure (r1 ∈ domain m) *
      R 0 r0 * R 3 r3 * … * R 8 r8 * R 9 err *
      pure (heap_inv (x0,Dot x y,x2,x3) (m,r0,r1,r2,r3,r4,r5,r6,r7,r8))) s
```

More Hoare triples

GC:

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  “ entire GC implementation ”
{ PC (pc + …) * HEAP (x0,x1,x2,x3,err) }
```

More Hoare triples

GC:

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  " entire GC implementation "
{ PC (pc + …) * HEAP (x0,x1,x2,x3,err) }
```

Allocation:

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  " allocation routine "
{ PC (pc + …) * HEAP (Dot x0 x1,x1,x2,x3,err)
  v
  PC err * true }
```

More Hoare triples

GC:

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  " entire GC implementation "
{ PC (pc + …) * HEAP (x0,x1,x2,x3,err) }
```

Allocation:

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  " allocation routine "
{ PC (pc + …) * HEAP (Dot x0 x1,x1,x2,x3,err)
  v
  PC err * true }
```

or, allowed to go to err when 'out of memory'

More Hoare triples

GC:

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  " entire GC implementation "
{ PC (pc + …) * HEAP (x0,x1,x2,x3,err) }
```

Allocation:

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  " allocation routine "
{ PC (pc + …) * HEAP (Dot x0 x1,x1,x2,x3,err)
  v
  PC err * true }
```

or, allowed to go to err when 'out of memory'

Update to flag:

```
{ PC pc * HEAP (x0,x1,x2,x3,err) * F _ }
  " test r0,3 "
{ PC (pc + 4) * HEAP (x0,x1,x2,x3,err) * F (isDot x0) }
```

More Hoare triples

GC:

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  " entire GC implementation "
{ PC (pc + …) * HEAP (x0,x1,x2,x3,err) }
```

Allocation:

```
{ PC pc * HEAP (x0,x1,x2,x3,err) }
  " allocation routine "
{ PC (pc + …) * HEAP (Dot x0 x1,x1,x2,x3,err)
  v
  PC err * true }
```

or, allowed to go to err when 'out of memory'

Update to flag:

this test sets flag to 'r0 && 3 == 0'

```
{ PC pc * HEAP (x0,x1,x2,x3,err) * F _ }
  " test r0,3 "
{ PC (pc + 4) * HEAP (x0,x1,x2,x3,err) * F (isDot x0) }
```

Reminder: Decompilation

Implementation:

1. compose Hoare triples along each path through code
2. apply loop rule, if applicable
3. read off function and precondition

Reminder: Decompilation

Implementation:

1. compose Hoare triples along each path through code
2. apply loop rule, if applicable
3. read off function and precondition

Example: step 1, composition of triples gives:

```
{ PC pc * R 0 r0 * R 1 r1 * F _ }
  “ FAC: cmp r0,0 ... ”
{ if r0 = 0 then
    PC (pc + 20) * R 0 r0 * R 1 r1 * F _
  else
    let r1 = r1 × r0 in
    let r0 = r0 - 1 in
      PC pc * R 0 r0 * R 1 r1 * F _ }
```

Reminder: Decompilation

Implementation:

1. compose Hoare triples along each path through code
2. apply loop rule, if applicable
3. read off function and precondition

Example: step 1, composition of triples gives:

```
{ PC pc * R 0 r0 * R 1 r1 * F _ }
  “ FAC: cmp r0,0 ... ”
{ if r0 = 0 then
    PC (pc + 20) * R 0 r0 * R 1 r1 * F _
  else
    let r1 = r1 × r0 in
    let r0 = r0 - 1 in
      PC pc * R 0 r0 * R 1 r1 * F _ }
```

What if decompilation used HEAP Hoare triples instead?

Decompilation of Lisp code

By supplying code with HEAP assertion-theorems to the decompiler we can make it extract functions over s-expressions.

Decompilation of Lisp code

By supplying code with HEAP assertion-theorems to the decompiler we can make it extract functions over s-expressions.

Example: composition of relevant HEAP assertion-theorems

```
… ⇒

{ PC pc * HEAP (x0,x1,x2,x3,err) * F _ }
  “ load r0,[r1] ... ”
{ let x0 = car x1 in
  let x0 = cdr x0 in
    if x2 = x0 then
      let x0 = car x1 in
      let x2 = cdr x0 in
        PC (pc + 24) * HEAP (x0,x1,x2,x3,err) * F _
    else
      let x1 = cdr x1 in
        PC pc * HEAP (x0,x1,x2,x3,err) * F _ }
```

Decompilation of Lisp code

By supplying code with HEAP assertion-theorems to the decompiler we can make it extract functions over s-expressions.

Example: composition of relevant HEAP assertion-theorems

```
… ⇒

{ PC pc * HEAP (x0,x1,x2,x3,err) * F _ }
  “ load r0,[r1] ... ”
{ let x0 = car x1 in
  let x0 = cdr x0 in
    if x2 = x0 then
      let x0 = car x1 in
      let x2 = cdr x0 in
        PC (pc + 24) * HEAP (x0,x1,x2,x3,err) * F _
    else
      let x1 = cdr x1 in
        PC pc * HEAP (x0,x1,x2,x3,err) * F _ }
```

Remainder of decompiler can operate as before.

Extracted functions

Function:

```
alist_lookup (x1,x2) =
  let x0 = car x1 in
  let x0 = cdr x0 in
    if x2 = x0 then
      let x0 = car x1 in
      let x2 = cdr x0 in
        (x0,x1,x2)
   else
      let x1 = cdr x1 in
        alist_lookup (x1,x2)
```

Extracted functions

Function:

```
alist_lookup (x1,x2) =
  let x0 = car x1 in
  let x0 = cdr x0 in
    if x2 = x0 then
      let x0 = car x1 in
      let x2 = cdr x0 in
        (x0,x1,x2)
   else
      let x1 = cdr x1 in
        alist_lookup (x1,x2)
```

Side condition:

```
alist_lookup_pre (x1,x2) =
  let c = isDot x1 in
  let x0 = car x1 in
  let c = isDot x0 ∧ c in
  let x0 = cdr x0 in
  let c = comparable x0 x2 ∧ c in
   if x2 = x0 then
      let c = isDot x1 ∧ c in
      let x0 = car x1 in
      let c = isDot x0 ∧ c in
      let x2 = cdr x0 in
        c
   else
      let c = isDot x1 ∧ c in
      let x1 = cdr x1 in
        alist_lookup_pre (x1,x2) ∧ c
```

Synthesis via decompilation

Function:

```
alist_lookup (x1,x2) =
  let x0 = car x1 in
  let x0 = cdr x0 in
    if x2 = x0 then
      let x0 = car x1 in
      let x2 = cdr x0 in
        (x0,x1,x2)
   else
      let x1 = cdr x1 in
        alist_lookup (x1,x2)
```

Synthesis via decompilation

Function:

```
alist_lookup (x1,x2) =
  let x0 = car x1 in
  let x0 = cdr x0 in
    if x2 = x0 then
      let x0 = car x1 in
      let x2 = cdr x0 in
        (x0,x1,x2)
    else
      let x1 = cdr x1 in
        alist_lookup (x1,x2)
```

Idea: start by writing this function!

Synthesis via decompilation

Function:

```
alist_lookup (x1,x2) =
  let x0 = car x1 in
  let x0 = cdr x0 in
    if x2 = x0 then
      let x0 = car x1 in
      let x2 = cdr x0 in
        (x0,x1,x2)
   else
      let x1 = cdr x1 in
        alist_lookup (x1,x2)
```

Idea: start by writing this function!

Method:

1. write s-expression function *f* in logic (tail-rec., names x0…x3)
2. generate code (without proof) based on function *f*
3. decompile code to some *f'*
4. automatically prove *f'* = *f*

Synthesis via decompilation

Function:

```
alist_lookup (x1,x2) =
  let x0 = car x1 in
  let x0 = cdr x0 in
    if x2 = x0 then
      let x0 = car x1 in
      let x2 = cdr x0 in
        (x0,x1,x2)
   else
      let x1 = cdr x1 in
        alist_lookup (x1,x2)
```

Idea: start by writing this function!

Method:

1. write s-expression function *f* in logic (tail-rec., names x0…x3)
2. generate code (without proof) based on function *f*
3. decompile code to some *f'*
4. automatically prove *f'* = *f*

the hard work is here

Synthesis via decompilation

Function:

```
alist_lookup (x1,x2) =
  let x0 = car x1 in
  let x0 = cdr x0 in
    if x2 = x0 then
      let x0 = car x1 in
      let x2 = cdr x0 in
        (x0,x1,x2)
   else
      let x1 = cdr x1 in
        alist_lookup (x1,x2)
```

Idea: start by writing this function!

Method:

1. write s-expression function *f* in logic (tail-rec., names x0…x3)
2. generate code (without proof) based on function *f*
3. decompile code to some *f'* — the hard work is here
4. automatically prove *f'* = *f* — light-weight proof, very simple

Synthesis via decompilation

Function:

```
alist_lookup (x1,x2) =
  let x0 = car x1 in
  let x0 = cdr x0 in
    if x2 = x0 then
      let x0 = car x1 in
      let x2 = cdr x0 in
        (x0,x1,x2)
   else
      let x1 = cdr x1 in
        alist_lookup (x1,x2)
```

Idea: start by writing this function!

Method:

style of approach is called *translation validation*

1. write s-expression function *f* in logic (tail-rec., names x0…x3)
2. generate code (without proof) based on function *f*
3. decompile code to some *f'*
4. automatically prove *f'* = *f*

the hard work is here

light-weight proof, very simple

A verified interpreter

Approach to construct verified interpreter:

1. define appropriate heap assertion
2. prove Hoare triples in terms of heap assertion
3. write interpreter as function using only operations from triples
4. synthesise code (decompile + equivalence proof)
5. prove that function implements high-level spec

A verified interpreter

Approach to construct verified interpreter:

1. define appropriate heap assertion
2. prove Hoare triples in terms of heap assertion
3. write interpreter as function using only operations from triples
4. synthesise code (decompile + equivalence proof)
5. prove that function implements high-level spec

must be tail rec.

A verified interpreter

Approach to construct verified interpreter:

1. define appropriate heap assertion
2. prove Hoare triples in terms of heap assertion
3. write interpreter as function using only operations from triples
4. synthesise code (decompile + equivalence proof)
5. prove that function implements high-level spec

must be tail rec.

can conveniently be based on small-step semantics definition from lecture 3!

Top-level theorem

If one proves that big-step evaluation implies termination of function, then:

$$(exp, a, fns) \Downarrow_{\mathrm{ev}} result \quad \Rightarrow$$

```
{ PC pc * HEAP (exp,a,fns,_,err) * F _ }
  “ entire interpreter implementation ”
{ PC (pc + …) * HEAP (result,_,_,_,err) * F _
  v PC err * true }
```

Top-level theorem

If one proves that big-step evaluation implies termination of function, then:

$(exp, a, fns) \Downarrow_{\text{ev}} result \quad \Rightarrow$

```
{ PC pc * HEAP (exp,a,fns,_,err) * F _ }
  " entire interpreter implementation "
{ PC (pc + …) * HEAP (result,_,_,_,err) * F _
  v PC err * true }
```

machine-code implementation will compute result according to big-step semantics

Top-level theorem

If one proves that big-step evaluation implies termination of function, then:

$$(exp, a, fns) \Downarrow_{\text{ev}} result \quad \Rightarrow$$

```
{ PC pc * HEAP (exp,a,fns,_,err) * F _ }
  “ entire interpreter implementation ”
{ PC (pc + …) * HEAP (result,_,_,_,err) * F _
  v PC err * true }
```

machine-code implementation will compute result according to big-step semantics

implementation allowed to jump to error pointer

Summary

Heap assertion

- formalises memory abstraction
- specifies details of concrete representation
- defined on top of Hoare assertions (MEM, R, etc.)
- certain code snippets implement abstract operations

Summary

Heap assertion

- formalises memory abstraction
- specifies details of concrete representation
- defined on top of Hoare assertions (MEM, R, etc.)
- certain code snippets implement abstract operations

Synthesis via decompilation

- decompiler can be set up to understand heap assertion
- synthesis via decompiler = translation validation
- plugs together verified snippets

Summary

Heap assertion

- formalises memory abstraction
- specifies details of concrete representation
- defined on top of Hoare assertions (MEM, R, etc.)
- certain code snippets implement abstract operations

Synthesis via decompilation

- decompiler can be set up to understand heap assertion
- synthesis via decompiler = translation validation
- plugs together verified snippets

Verified interpreter

- can be constructed by synthesis of small-step-like tail-rec. fun.