Following instruction sequences are rejected by our type system.

$\mathsf{l1} : \mathsf{r1} := \mathsf{l2}; \mathsf{r3} := \mathsf{r2} + \mathsf{1}; \ldots$

$\mathsf{l3} : \mathsf{r1} := \mathsf{5};\ \mathsf{jump}\ \mathsf{r1}$

Following instruction sequences are rejected by our type system.

$\mathsf{l1} : \mathsf{r1} := \mathsf{l2}; \mathsf{r3} := \mathsf{r2} + \mathsf{1}; \ldots$

$\mathsf{l3} : \mathsf{r1} := \mathsf{5};\ \mathsf{jump}\ \mathsf{r1}$

- We haven't discussed how to check if a machine is well typed. Alternative: use proof carrying code.

Following instruction sequences are rejected by our type system.

$\mathsf{l1} : \mathsf{r1} := \mathsf{l2}; \mathsf{r3} := \mathsf{r2} + \mathsf{1}; \ldots$

$\mathsf{l3} : \mathsf{r1} := \mathsf{5};\ \mathsf{jump}\ \mathsf{r1}$

- We haven't discussed how to check if a machine is well typed. Alternative: use proof carrying code.
- It is straightforward to translate TAL-0 programs to code for some real processor.

  If the TAL-0 program is well-typed then the translated code will behave properly.

Following instruction sequences are rejected by our type system.

$\mathsf{l1} : \mathsf{r1} := \mathsf{l2}; \mathsf{r3} := \mathsf{r2} + \mathsf{1}; \ldots$

$\mathsf{l3} : \mathsf{r1} := \mathsf{5};\ \mathsf{jump}\ \mathsf{r1}$

- We haven't discussed how to check if a machine is well typed. Alternative: use proof carrying code.
- It is straightforward to translate TAL-0 programs to code for some real processor.

  If the TAL-0 program is well-typed then the translated code will behave properly.

  . . . for that we of course need to prove type safety for TAL-0 . . .

Type safety for TAL-0

"well typed machines do not get stuck"

Progress: If $\vdash M$ then there is some M' such that $M \rightarrow M'$.

Preservation: If $\vdash M$ and $M \rightarrow M'$ then $\vdash M'$.

An extension: TAL-1

We now also deal with memory safety.

Besides registers, we now have a potentially infinite memory, stack, pointers, and facilities for allocating space for data.

Already expressive enough for implementing simple programs from high level languages.

Memory safety: no reads to or writes from illegal memory locations.

Examples of new kinds of instructions

- $\mathsf{r1} := \mathsf{Mem}[\mathsf{r2} + \mathsf{4}]$

  $\mathsf{r2}$ stores a pointer. We access the 4th location past the corresponding memory location. The value there is loaded in $\mathsf{r1}$.

Examples of new kinds of instructions

- $\mathsf{r1} := \mathsf{Mem}[\mathsf{r2} + \mathsf{4}]$

  $\mathsf{r2}$ stores a pointer. We access the 4th location past the corresponding memory location. The value there is loaded in $\mathsf{r1}$.

- $\mathsf{Mem}[\mathsf{r2} + \mathsf{4}] := \mathsf{r1}$

  The reverse store operation.

Examples of new kinds of instructions

- $\mathsf{r1} := \mathsf{Mem}[\mathsf{r2} + \mathsf{4}]$

  r2 stores a pointer. We access the 4th location past the corresponding memory location. The value there is loaded in r1.

- $\mathsf{Mem}[\mathsf{r2} + \mathsf{4}] := \mathsf{r1}$

  The reverse store operation.

- $\mathsf{r1} := \mathsf{malloc}\ \mathsf{10}$

  allocate 10 words on the heap, and store corresponding pointer in r1.

Examples of new kinds of instructions

- $\mathsf{r1} := \mathsf{Mem}[\mathsf{r2} + \mathsf{4}]$

  r2 stores a pointer. We access the 4th location past the corresponding memory location. The value there is loaded in r1.

- $\mathsf{Mem}[\mathsf{r2} + \mathsf{4}] := \mathsf{r1}$

  The reverse store operation.

- $\mathsf{r1} := \mathsf{malloc}\ \mathsf{10}$

  allocate 10 words on the heap, and store corresponding pointer in r1.

- $\mathsf{salloc}\ \mathsf{10}$

  allocate 10 words on the stack (and update stack pointer)

Example code.

```
r1 := malloc 5;        // allocate 5 words on heap
Mem[r1] := 10;         // store data in the first word
Mem[r1 + 1] := 20;     // store data in the first word
r2 := Mem[r1]          // load 10 into r2
```

Example code.

```
r1 := malloc 5;          // allocate 5 words on heap
Mem[r1] := 10;           // store data in the first word
Mem[r1 + 1] := 20;       // store data in the first word
r2 := Mem[r1]            // load 10 into r2
```

The following code has no well-defined behavior.

```
r1 := malloc 5;    // allocate 5 words on heap
r2 := malloc 5;    // allocate 5 words on heap
r3 := r1 + r2      // add the two pointers
```

Example code.

```
r1 := malloc 5;           // allocate 5 words on heap
Mem[r1] := 10;            // store data in the first word
Mem[r1 + 1] := 20;        // store data in the first word
r2 := Mem[r1]             // load 10 into r2
```

The following code has no well-defined behavior.

```
r1 := malloc 5;     // allocate 5 words on heap
r2 := malloc 5;     // allocate 5 words on heap
r3 := r1 + r2       // add the two pointers
```

Hence for type safety, we should at least have a different type for pointers.

Further the type system should distinguish between pointers to different types of data.

```
r1 := malloc 5;
Mem[r1] := 9;
r2 := Mem[r1]     // r1 stores a pointer, hence this is ok
 jump r2          // not ok, since r1 was a pointer to an integer
```

Hence the type-system should deal with types like $\mathsf{ptr}(\mathsf{Int})$, $\mathsf{ptr}(\mathsf{Code}(\Gamma))$, $\mathsf{ptr}(\mathsf{ptr}(\mathsf{Int}))$, ...

```
                    // currently r1 : ptr(Code(...))
r3 := 5;
Mem[r1] := r3;      // now r1 : ptr(Int)
r4 := Mem[r1];      // r4 : Int
 jump r4            // of course ill-typed
```

Hence type of a register should be updated after a store through it.

Aliasing problem

Should the following be well typed?

$$
\begin{array}{ll}
 & // \text{ currently } \mathsf{r1} : \mathsf{ptr}(\mathsf{Code}(\ldots)), \mathsf{r2} : \mathsf{ptr}(\mathsf{Code}(\ldots)) \\
\mathsf{r3} := \mathsf{5}; & \\
\mathsf{Mem}[\mathsf{r1}] := \mathsf{r3}; & // \text{ now } \mathsf{r1} : \mathsf{ptr}(\mathsf{Int}) \\
\mathsf{r4} := \mathsf{Mem}[\mathsf{r2}]; & // \text{ load through } \mathsf{r2}.\ \mathsf{r4} :??? \\
\mathsf{jump}\ \mathsf{r4} & // \text{ is this well-typed???}
\end{array}
$$

Aliasing problem

Should the following be well typed?

```
                    // currently $\mathsf{r1} : \mathsf{ptr}(\mathsf{Code}(\ldots)), \mathsf{r2} : \mathsf{ptr}(\mathsf{Code}(\ldots))$
 r3 := 5;
 Mem[r1] := r3;     // now $\mathsf{r1} : \mathsf{ptr}(\mathsf{Int})$
 r4 := Mem[r2];     // load through r2. r4 :???
 jump r4            // is this well-typed???
```

Answer: depends on whether r1 and r2 point to the same location (aliasing).

Aliasing problem

Should the following be well typed?

```
                        // currently r1 : ptr(Code(...)), r2 : ptr(Code(...))
 r3 := 5;
 Mem[r1] := r3;         // now r1 : ptr(Int)
 r4 := Mem[r2];         // load through r2. r4 :???
  jump r4               // is this well-typed???
```

Answer: depends on whether r1 and r2 point to the same location (aliasing).

Problem: how should the type system keep track of aliasing?

Solution: have two kinds of memory locations.

Shared pointers: support aliasing. Different type of data cannot be written.

Unique pointers: no aliasing. Different kind of data can be written. Useful for allocating and initializing shared data structures, and for stack frames.

The instruction

$$\mathsf{commit}\ r_d$$

declares a pointer to be shared, its type cannot change now.

The TAL-1 syntax: we make the following extensions to the TAL-0 syntax.

$r ::=$	registers
$\mathsf{r1} \mid \ldots \mid \mathsf{rk} \mid \mathsf{sp}$	ordinary registers and stack pointer
$\iota ::=$	instructions
$\ldots$	mov/add/if-jump
$r_d := \mathsf{Mem}[r_s + n]$	load from memory
$\mathsf{Mem}[r_d + n] := r_s$	store to memory
$r_d := \mathsf{malloc}\ n$	allocate n heap words
$\mathsf{commit}\ r_d$	make the pointer shared
$\mathsf{salloc}\ n$	allocate n stack words
$\mathsf{sfree}\ n$	free n stack words

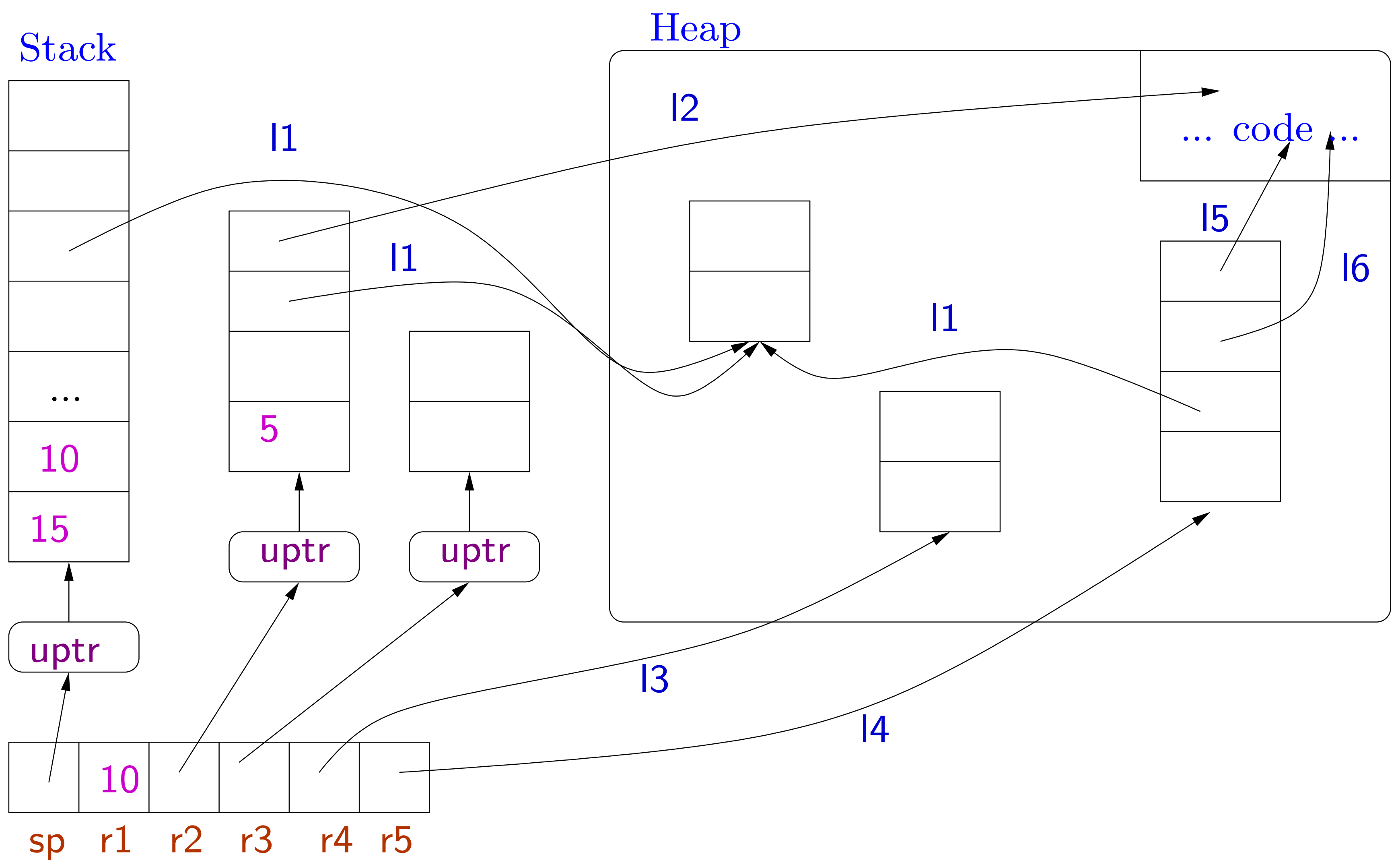
$\nu ::=$		operands
	r	registers
	n	integers
	l	code or shared data pointers
	$\mathsf{uptr}(h)$	unique data pointers
$h ::=$		heap values
	I	instruction sequences
	$\langle \nu_1, \ldots, \nu_n \rangle$	tuples

Instr. sequences I are as in TAL-0: list of instructions followed by a jump

Values are operands other than registers. Heaps map labels l to heap values h.

Register files and abstract machine states are defined as for TAL-0.

The TAL-1 abstract machine: Unique data values are not stored in the heap.



TAL-1 evaluation rules

We fix a constant $\mathsf{MaxStack}$: the maximum allowed size of the stack.

All TAl-0 evaluation rules remain the same except the (E-Mov) rule.

This rule now needs to ensure that unique pointers are not copied.

$$\frac{\hat{R}(\nu) \neq \mathsf{uptr}(h)}{(H, R, r_d := \nu; I) \rightarrow (H, R \oplus \{r_d \mapsto \hat{R}(\nu)\}, I)} \text{(E-Mov1)}$$

The $\hat{R}$ function is as for TAL-0. Further we have $\hat{R}(\mathsf{uptr}(h)) = \mathsf{uptr}(h)$.

If $\hat{R}(\nu)$ is $\mathsf{uptr}(h)$ then the machine gets stuck.

TAL-1 evaluation rules

We fix a constant $\mathsf{MaxStack}$: the maximum allowed size of the stack.

All TAl-0 evaluation rules remain the same except the (E-Mov) rule.

This rule now needs to ensure that unique pointers are not copied.

$$\frac{\hat{R}(\nu) \neq \mathsf{uptr}(h)}{(H, R, r_d := \nu; I) \to (H, R \oplus \{r_d \mapsto \hat{R}(\nu)\}, I)} \text{(E-Mov1)}$$

The $\hat{R}$ function is as for TAL-0. Further we have $\hat{R}(\mathsf{uptr}(h)) = \mathsf{uptr}(h)$.

If $\hat{R}(\nu)$ is $\mathsf{uptr}(h)$ then the machine gets stuck.

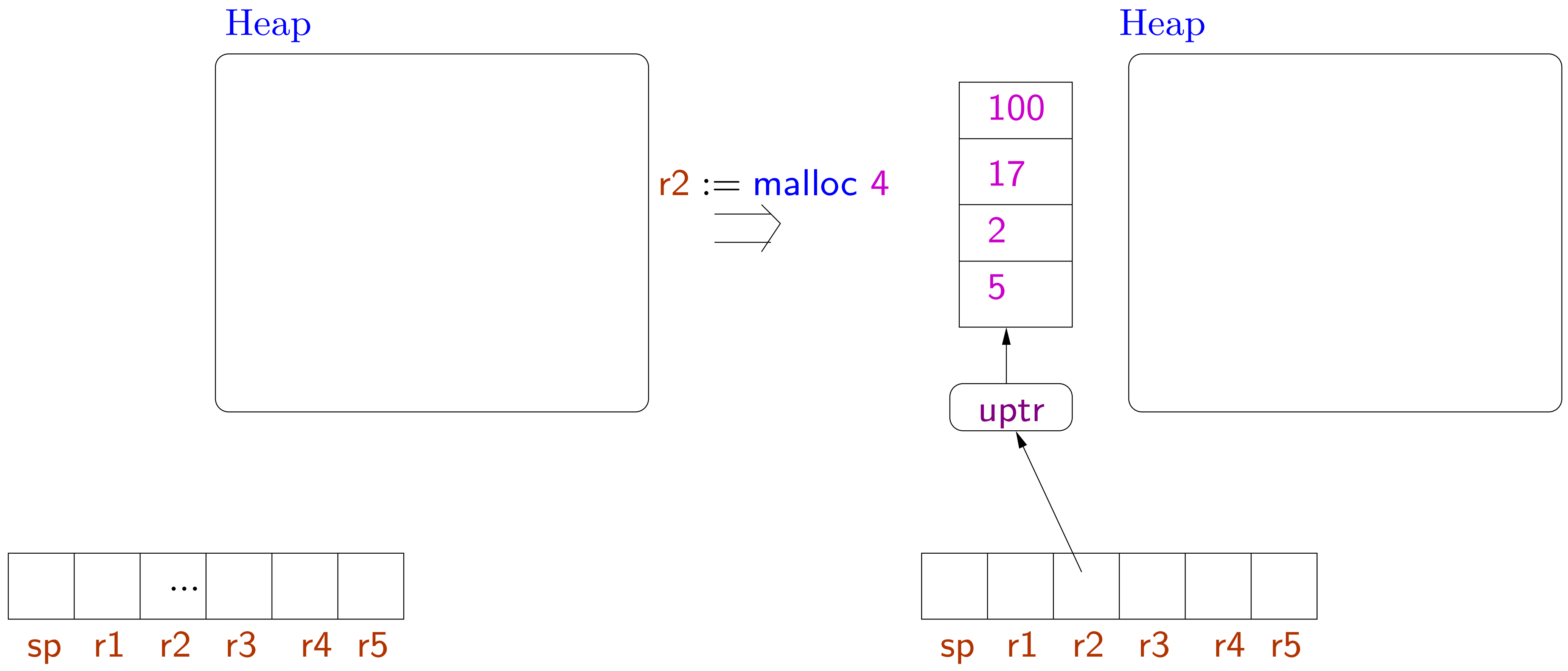
The other evaluation rules of TAL-0 are unmodified. We now add new rules for the new instructions …

Allocation generates a unique pointer

$$(H, R, r_d := \mathsf{malloc}\ n; I) \rightarrow (H, R \oplus \{r_d \mapsto \mathsf{uptr}\langle m_1, \ldots, m_n \rangle\}, I) \qquad \text{(E-Malloc)}$$

- A unique pointer to a tuple of n words is created and stored in the destination register.
- The initial values in the words are arbitrary integers $m_1, \ldots, m_n$ (uninitialized values)
- Typically we would make the pointer shared once the words have been initialized.
- malloc instruction takes a constant as argument. Useful for implementing tuples, records, etc but not yet for variable sized arrays.

Allocation



Heap
r2 := malloc 4
100
17
2
5
uptr
Heap
...
sp
r1
r2
r3
r4
r5
sp
r1
r2
r3
r4
r5

Examples The following code will lead to stuck states.

- copying of unique pointers:

$$\ldots \mathsf{r1} := \mathsf{malloc}\ \mathsf{5}; \mathsf{r2} := \mathsf{r1}; \ldots$$

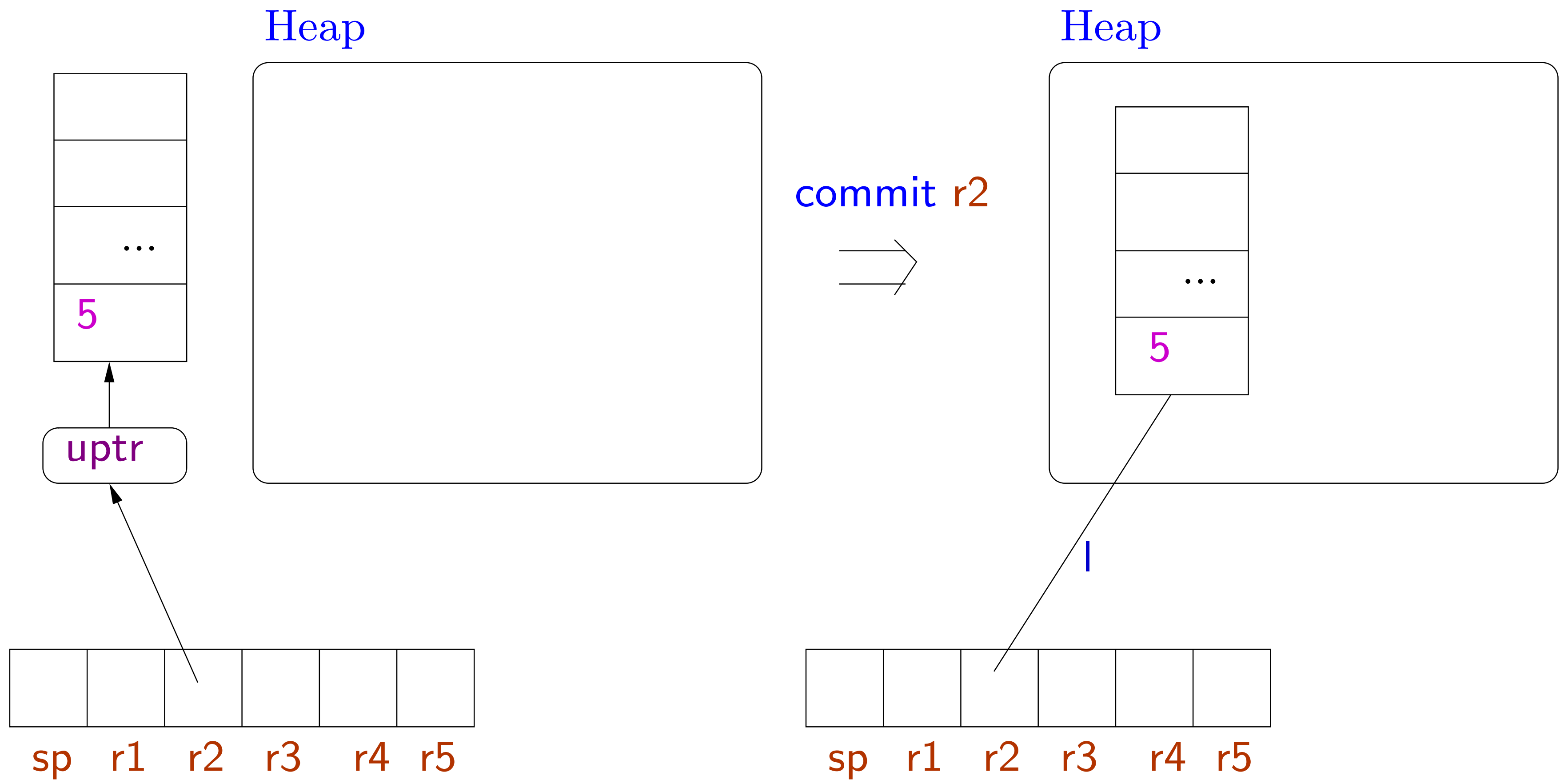
- using unique pointers in place of integers

$$\ldots \mathsf{r1} := \mathsf{malloc}\ \mathsf{5}; \mathsf{if}\ \mathsf{r1}\ \mathsf{jump}\ \mathsf{l}; \ldots$$

Declaring a pointer to be shared

$$\frac{r_d \neq \mathsf{sp} \quad R(r_d) = \mathsf{uptr}(h) \quad l \notin dom(H)}{(H, R, \mathsf{commit}\ r_d; I) \rightarrow (H \oplus \{l \mapsto h\}, R \oplus \{r_d \mapsto l\}, I)}\ \text{(E-Commit)}$$

- The stack is always a unique data value.
- commit moves the unique data in the heap (i.e. it is now considered shared data)
- A fresh label is associated with the data and is stored in the destination register.



l is a completely fresh label.

Loading and storing

Loading shared data

$$\frac{R(r_s) = l \quad H(l) = \langle \nu_0, \ldots, \nu_n, \ldots, \rangle}{(H, R, r_d := \mathsf{Mem}[r_s + n]; I) \longrightarrow (H, R \oplus \{r_d \mapsto \nu_n\}, I)} \text{(E-Ld-S)}$$

Loading and storing

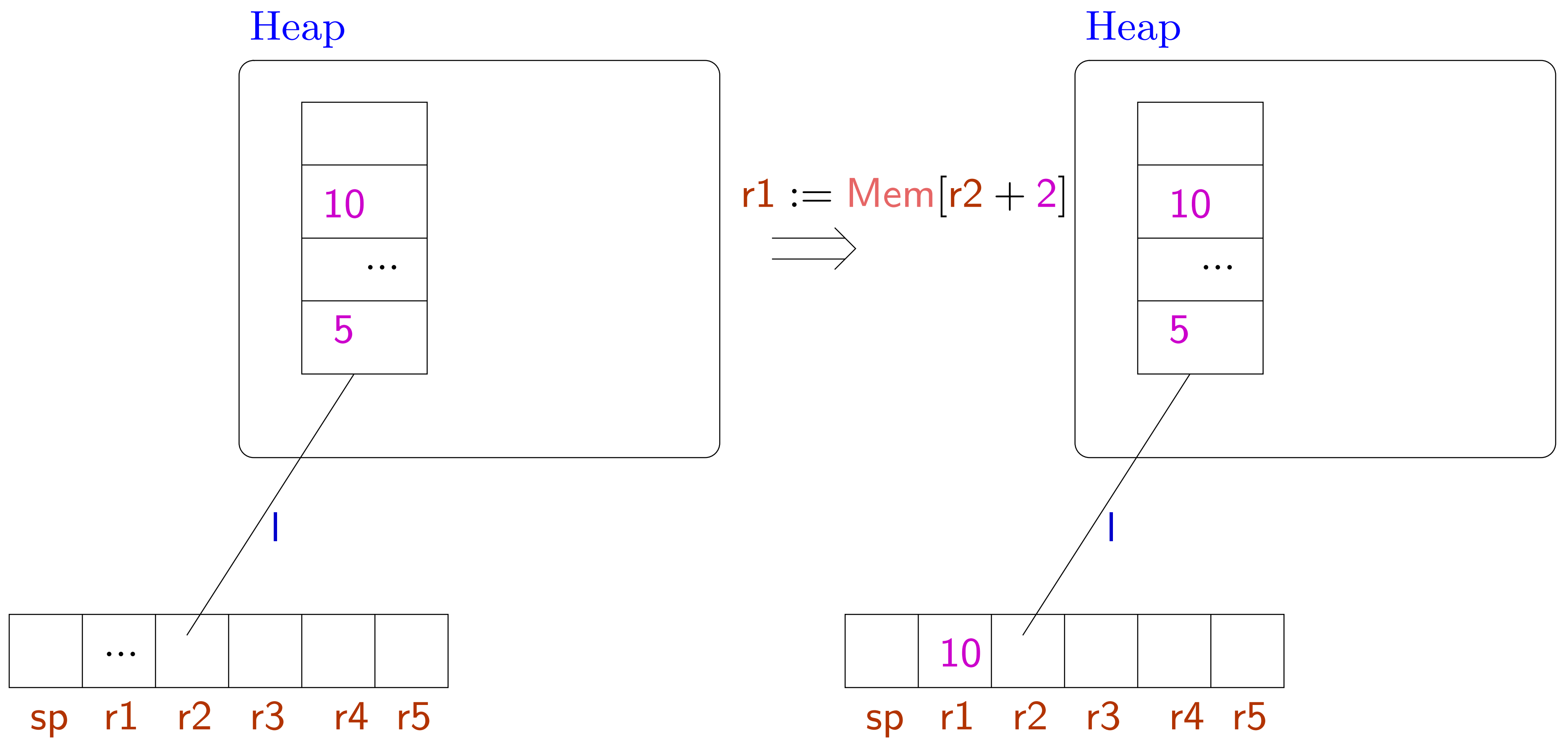
Loading shared data

$$\frac{R(r_s) = l \quad H(l) = \langle \nu_0, \ldots, \nu_n, \ldots, \rangle}{(H, R, r_d := \mathsf{Mem}[r_s + n]; I) \rightarrow (H, R \oplus \{r_d \mapsto \nu_n\}, I)} \text{(E-Ld-S)}$$

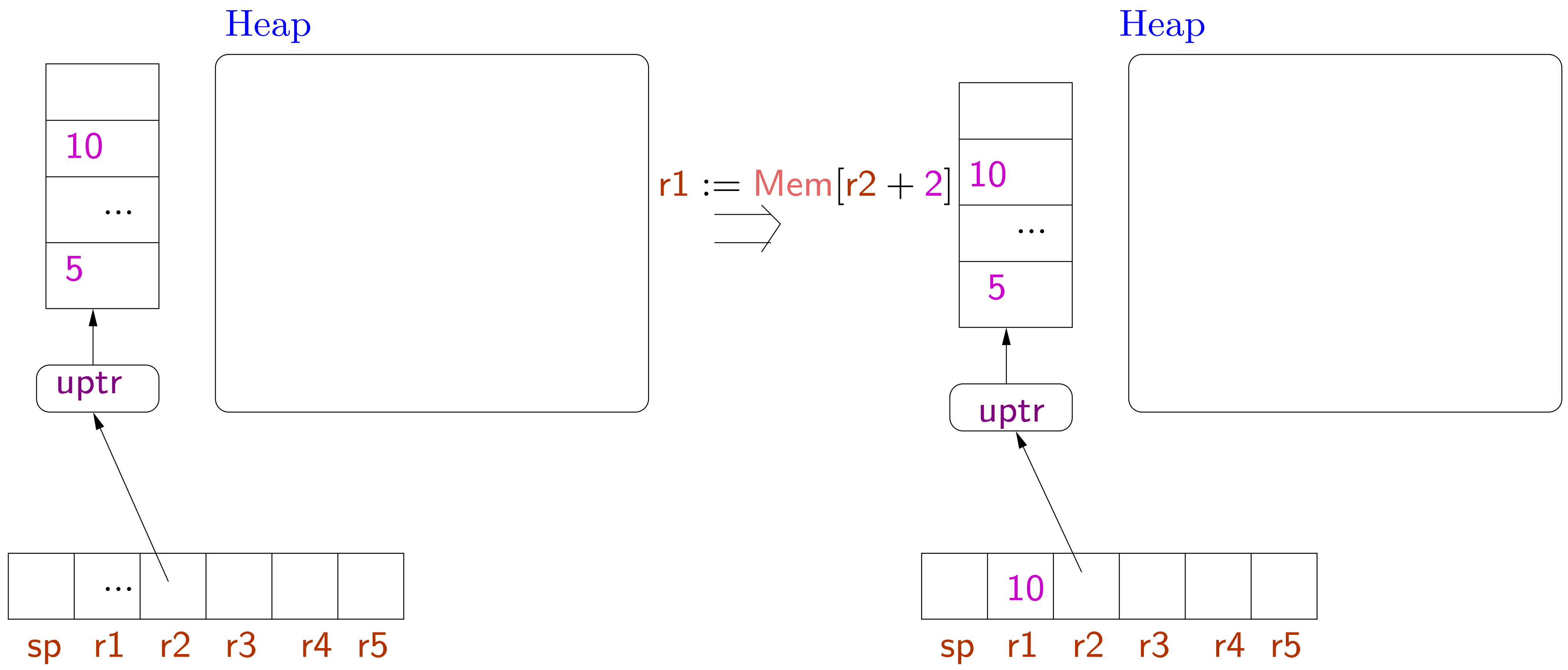
Loading unique data

$$\frac{R(r_s) = \mathsf{uptr}\langle \nu_0, \ldots, \nu_n, \ldots, \rangle}{(H, R, r_d := \mathsf{Mem}[r_s + n]; I) \rightarrow (H, R \oplus \{r_d \mapsto \nu_n\}, I)} \text{(E-Ld-U)}$$

Loading shared data



Loading unique data



Storing shared data

$$\frac{R(r_d) = l \quad H(l) = \langle \nu_0, \ldots, \nu_n, \ldots, \rangle \quad R(r_s) = \nu \quad \nu \neq \mathsf{uptr}(h)}{(H, R, \mathsf{Mem}[r_d + n] := r_s; I) \to (H \oplus \{l \mapsto \langle \nu_0, \ldots, \nu, \ldots, \rangle\}, R, I)} \text{(E-St-S)}$$

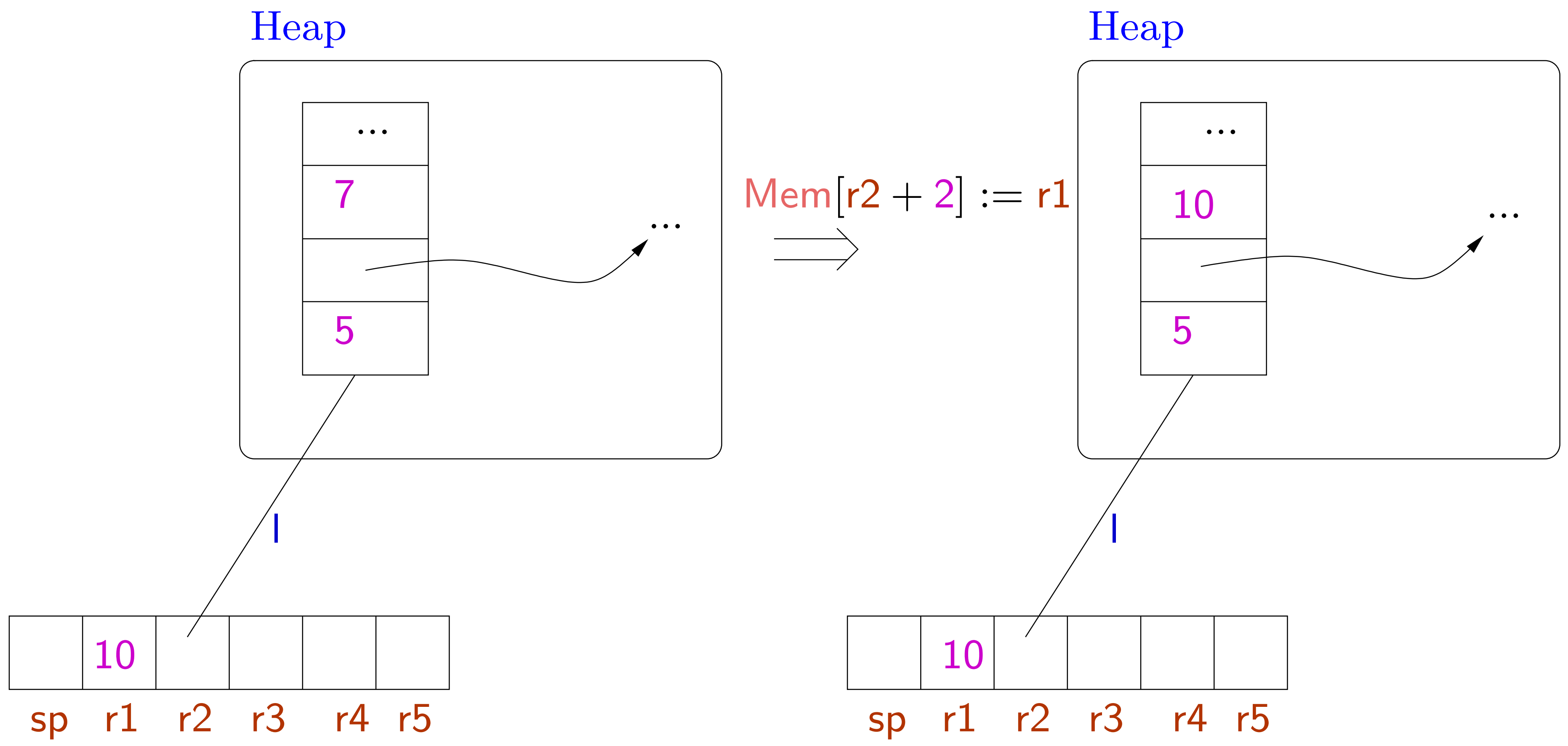
Storing shared data

$$\frac{R(r_d) = l \quad H(l) = \langle \nu_0, \ldots, \nu_n, \ldots, \rangle \quad R(r_s) = \nu \quad \nu \neq \mathsf{uptr}(h)}{(H, R, \mathsf{Mem}[r_d + n] := r_s; I) \rightarrow (H \oplus \{l \mapsto \langle \nu_0, \ldots, \nu, \ldots, \rangle\}, R, I)} \text{(E-St-S)}$$

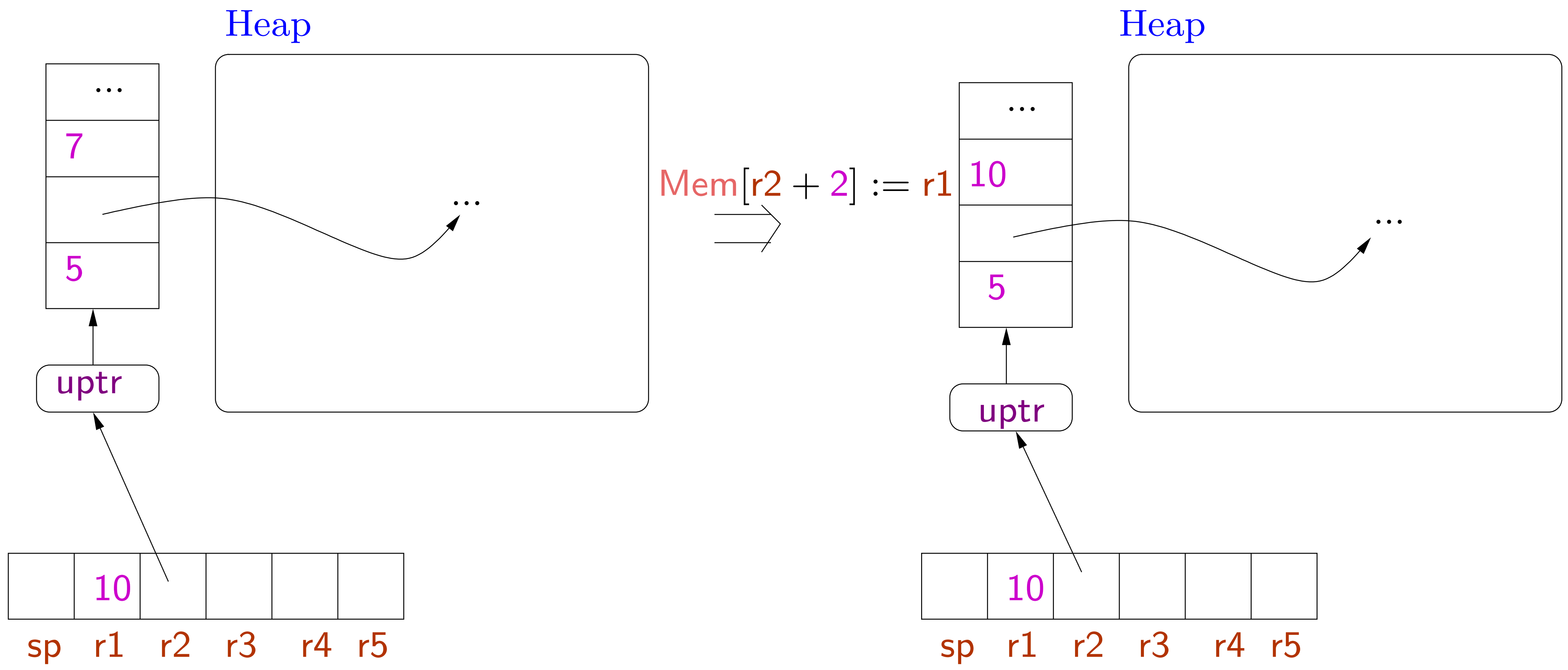
Storing unique data

$$\frac{R(r_d) = \mathsf{uptr}\langle \nu_0, \ldots, \nu_n, \ldots, \rangle \quad R(r_s) = \nu \quad \nu \neq \mathsf{uptr}(h)}{(H, R, \mathsf{Mem}[r_d + n] := r_s; I) \rightarrow (H, R \oplus \{r_d \mapsto \mathsf{uptr}\langle \nu_0, \ldots, \nu, \ldots, \rangle\}, I)} \text{(E-St-U)}$$

Storing shared data



Storing unique data



Example Allocating space, initializing data, and making it shared.

```
l :   r1 := malloc 3;
      r3 := l;
      r4 := 7;
      Mem[r1] = r3;
      Mem[r1 + 1] = r4;
      commit r1;
      r2 := r1;              // now the pointer can be aliased
      r4 := r4 + 6;
      Mem[r2 + 1] := r4;     // this is ok (should be well-typed)
      Mem[r2 + 1] := r3;     // this is not ok
```

This is also ok.

```
l :  r1 := malloc 3;
     r3 := l;
     r4 := 7;
     Mem[r1] = r4;    //r1 : uptr(Int, ...)
     Mem[r1] = r3;    //r1 : uptr(Code(...), ...)
     ...
     commit r1;
```

Type of data can change before being declared to be shared.

Allocation on the stack

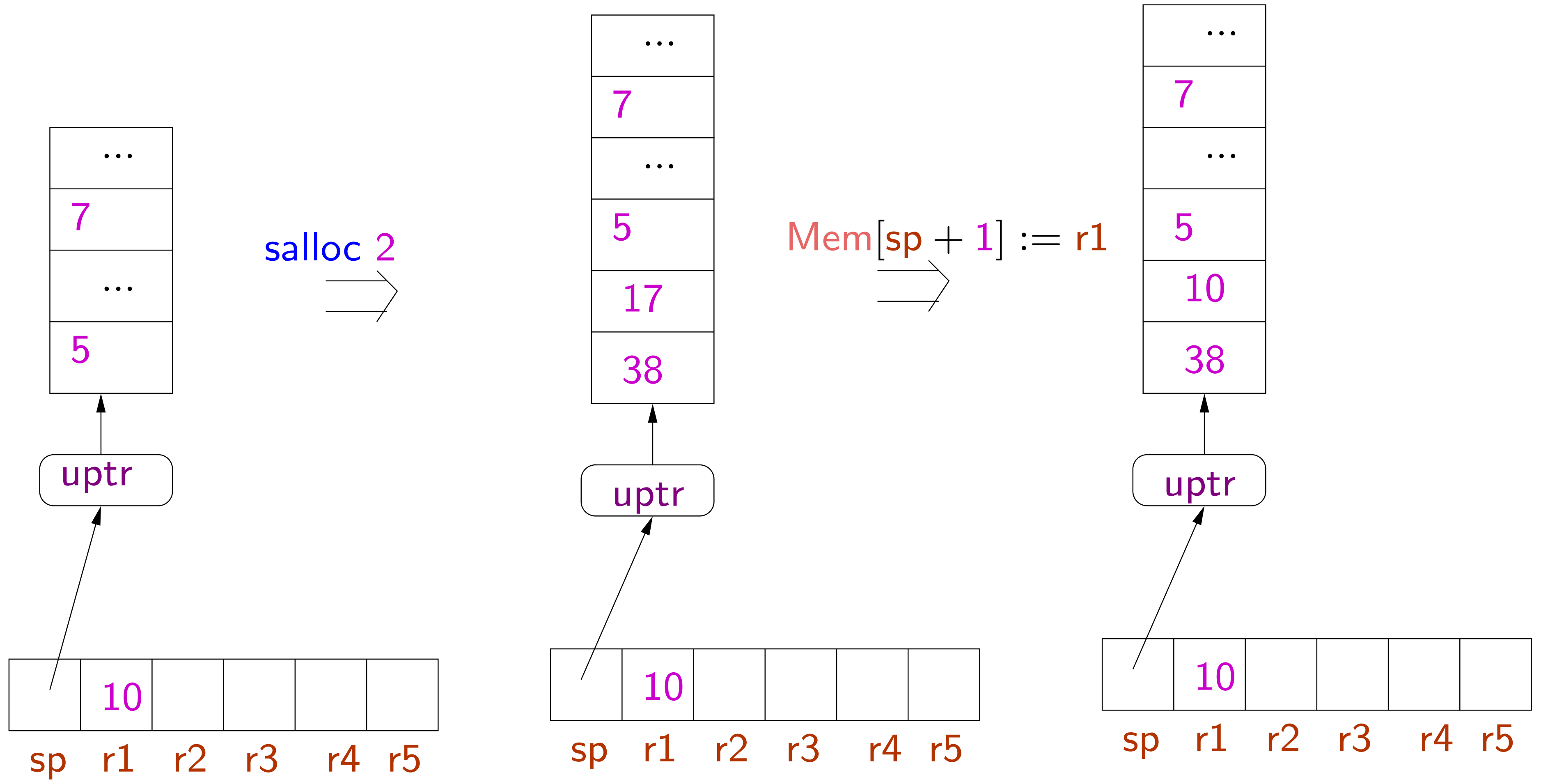
$$\frac{R(\mathsf{sp}) = \mathsf{uptr}\langle \nu_0, \ldots, \nu_p \rangle \quad p + n \leq \mathsf{MaxStack}}{(H, R, \mathsf{salloc}\ n; I) \to (H, R \oplus \{\mathsf{sp} \mapsto \mathsf{uptr}\langle m_1, \ldots, m_n, \nu_0, \ldots, \nu_p \rangle\}, I)}\ \text{(E-Salloc)}$$

- The stack is a unique data.
- Instead of allocating a new tuple, we extend the existing stack
- Arbitrary integers (uninitialized values) are added at the top of the stack.
- Stack overflow leads to stuck state.
- positive indexing for stack as for other data tuples.

Deallocating space from the stack

$$\frac{R(\mathsf{sp}) = \mathsf{uptr}\langle \nu_1', \ldots, \nu_n', \nu_0, \ldots, \nu_p \rangle}{(H, R, \mathsf{sfree}\ n; I) \rightarrow (H, R \oplus \{\mathsf{sp} \mapsto \mathsf{uptr}\langle \nu_0, \ldots, \nu_p \rangle\}, I)} \text{(E-Sfree)}$$

- Stack underflow leads to a stuck state: the stack should have at least n elements before the sfree instruction.


...
7
...
5
uptr
10
sp
r1
r2
r3
r4
r5
salloc 2
...
7
...
5
17
38
uptr
10
sp
r1
r2
r3
r4
r5
Mem[sp + 1] := r1
...
7
...
5
10
38
uptr
10
sp
r1
r2
r3
r4
r5

- No call/return instructions in the language.
- These are simulated using the jump instruction: e.g. saving/restoring return addresses are done explicitly.
- Allows modifications in calling conventions (passing arguments and return address on stack or in registers, tail recursion, . . .)
- For this we focus on a more primitive set of type constructors.
- In contrast, the JVM language has notions of procedures and procedure calls hardwired into the language. Any modification (e.g. adding tail recursion) requires modifications in the abstract machine and the type system.

Translations from high level languages to TAL-1

TAL-1 is expressive enough to implement simple subsets of high level languages.

Example C Code

```
int fib (int x) {
   if (x == 0) return 0; else
   if (x == 1) return 1; else
  return (fib (n−1) + fib (n−2));
}
```

We choose the following calling conventions for our example.

- Caller pushes arguments on the stack.
- Caller puts return address in r3.
- Callee pops arguments from the stack.
- Callee returns the result in r1.
- Register r2 is freely available for intermediate computations.

```
fib :   r2 := Mem[sp];            // r2 := x
        if r2 jump ret0;
        r2 := r2 + −1;            // r2 := x − 1
        if r2 jump ret1;
        salloc 2;
        Mem[sp + 1] := r3;        // save old return address
        Mem[sp] := r2;            // push x − 1 on stack
        r3 := cont1;              // new return address
        jump fib                  // r1 := fib(x − 1)
```

```
ret0 :   r1 := 0;     // return value
         sfree 1;     // pop argument
          jump r3     // return
```

```
ret1 :   r1 := 1;
         sfree 1;
          jump r3
```

```
ret0 :  r1 := 0;     // return value
        sfree 1;     // pop argument
        jump r3      // return
```

```
ret1 :  r1 := 1;
        sfree 1;
        jump r3
```

```
cont1 :  salloc 2;
         Mem[sp + 1] := r1;   // save fib(x − 1)
         r2 := Mem[sp + 3];   // r2 := x
         r2 := r2 + −2;       // r2 := x − 2
         Mem[sp] := r2;       // push x − 2 on stack
         r3 := cont2;         // push return address
         jump fib             // r1 = fib(x − 2)
```

```
cont2 :   r2 := Mem[sp];          // r2 := fib(x − 1)
          r1 := r1 + r2;          // r1 := fib(x − 2) + fib(x − 1)
          r3 := Mem[sp + 1];      // restore old return address
          sfree 3;
           jump r3
```

Towards a TAL-1 type system

How to distinguish ”good” programs from ”bad” programs?

As discussed, we need types

$\mathsf{ptr}(\sigma)$	unique pointer type
$\mathsf{uptr}(\sigma)$	shared pointer type

where σ is an allocated type, i.e. type for allocated data.

The instruction $\mathsf{r1} := \mathsf{malloc}\ 3$ makes the register $\mathsf{r1}$ to be of type $\mathsf{uptr}\langle \mathsf{Int}, \mathsf{Int}, \mathsf{Int}\rangle$.

The instruction $\mathsf{commit}\ \mathsf{r2}$ transforms the type of register $\mathsf{r2}$ from $\mathsf{uptr}(\sigma)$ to $\mathsf{ptr}(\sigma)$.

Consider the fib example again.

Initially sp should point to a stack having Int at the top.

However the rest of the stack could be arbitrarily large and have elements of arbitrary type.

Consider the fib example again.

Initially sp should point to a stack having Int at the top.

However the rest of the stack could be arbitrarily large and have elements of arbitrary type.

First idea: use a type similar to Top, to represent tuples of ”any” type.

Further this should type should also represent tuples of any length.

Suppose we choose a type Top′ for this.

Then fib would expect sp to have type $\langle \mathsf{Int}, \mathsf{Top}' \rangle$, representing a stack with an integer at the top and any number of other things below.

Hence we should expect:

$\mathsf{fib} : \mathsf{Code}\{\mathsf{sp} : \mathsf{uptr}\langle \mathsf{Int}, \mathsf{Top}' \rangle, \mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}, \mathsf{r3} : \mathsf{Code}(\Gamma)\}$.

What should be Γ?

At the end of computation, we have $\mathsf{r1} : \mathsf{Int}$, $\mathsf{sp} : \mathsf{uptr}(\mathsf{Top}')$, and we jump to the label l contained in $\mathsf{r3}$.

Hence we should expect:

$\Gamma = \{\mathsf{sp} : \mathsf{uptr}(\mathsf{Top}'), \mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Top}, \mathsf{r3} : \mathsf{Top}\}$.

Then fib would expect sp to have type $\langle \mathsf{Int}, \mathsf{Top}' \rangle$, representing a stack with an integer at the top and any number of other things below.

Hence we should expect:

$\mathsf{fib} : \mathsf{Code}\{\mathsf{sp} : \mathsf{uptr}\langle \mathsf{Int}, \mathsf{Top}' \rangle, \mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}, \mathsf{r3} : \mathsf{Code}(\Gamma)\}.$

What should be Γ?

At the end of computation, we have $\mathsf{r1} : \mathsf{Int}$, $\mathsf{sp} : \mathsf{uptr}(\mathsf{Top}')$, and we jump to the label l contained in $\mathsf{r3}$.

Hence we should expect:

$\Gamma = \{\mathsf{sp} : \mathsf{uptr}(\mathsf{Top}'), \mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Top}, \mathsf{r3} : \mathsf{Top}\}.$

But we are forgetting the relationship between the types of values on the stack at the beginning and at the end!

Solution: use type variables to state such equalities.

Hence with fib we will associate the type

$$\forall \mathsf{s} \cdot \mathsf{Code}\{\mathsf{sp} : \mathsf{uptr}\langle \mathsf{Int}, \mathsf{s}\rangle, \mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top},$$
$$\mathsf{r3} : \mathsf{Code}\{\mathsf{sp} : \mathsf{uptr}(\mathsf{s}), \mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Top}, \mathsf{r3} : \mathsf{Top}\}\}$$

where s is an allocated type variable i.e. representing an arbitrary length of allocated memory.

This expresses the constraint that the code pointed to by r3 should expect the same type of stack that is below the argument of fib.

The universal quantifier helps to distinguish occurrences of the variable s elsewhere.