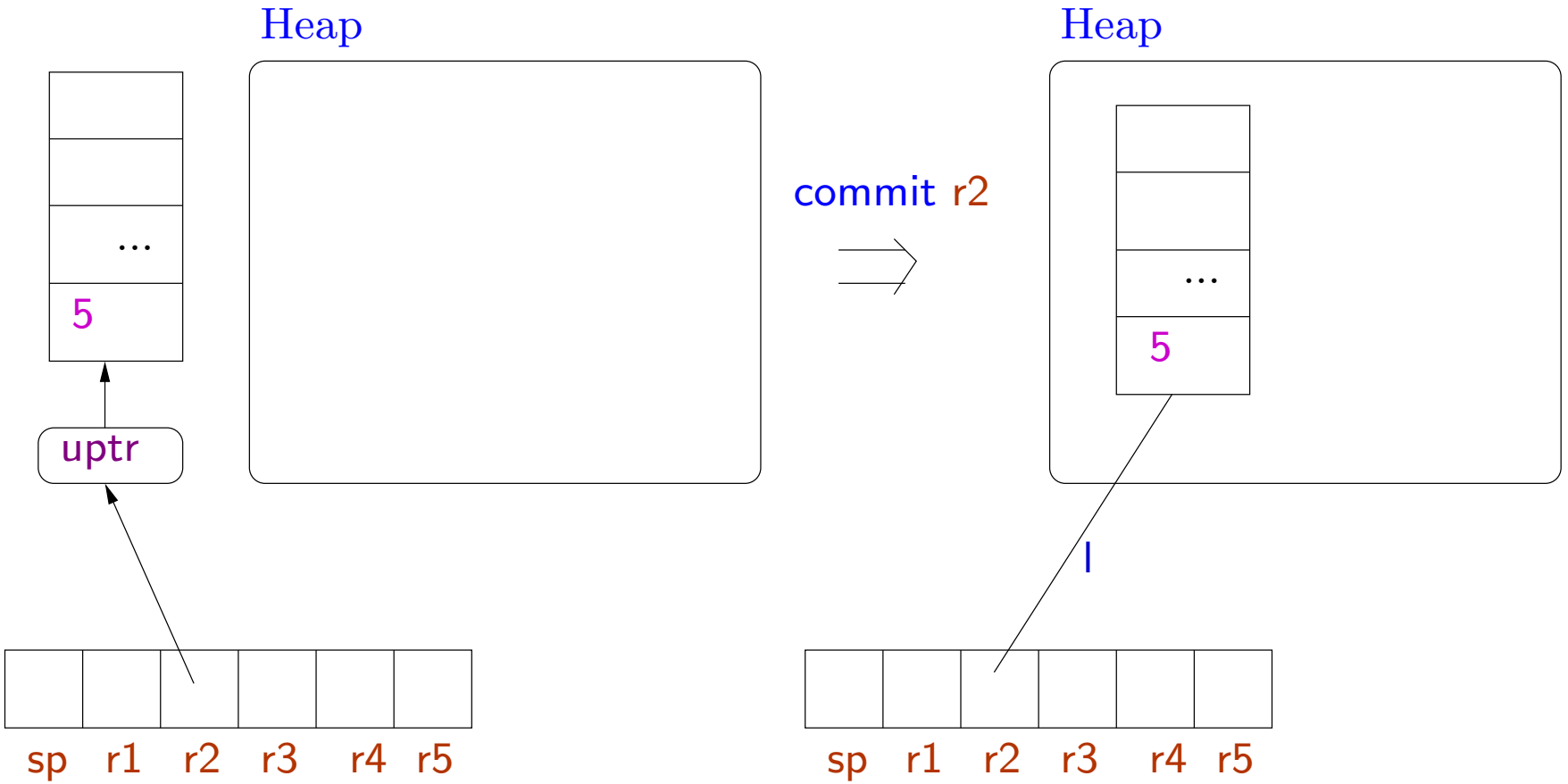


l is a completely fresh label.

Loading and storing

Loading shared data

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

Loading and storing

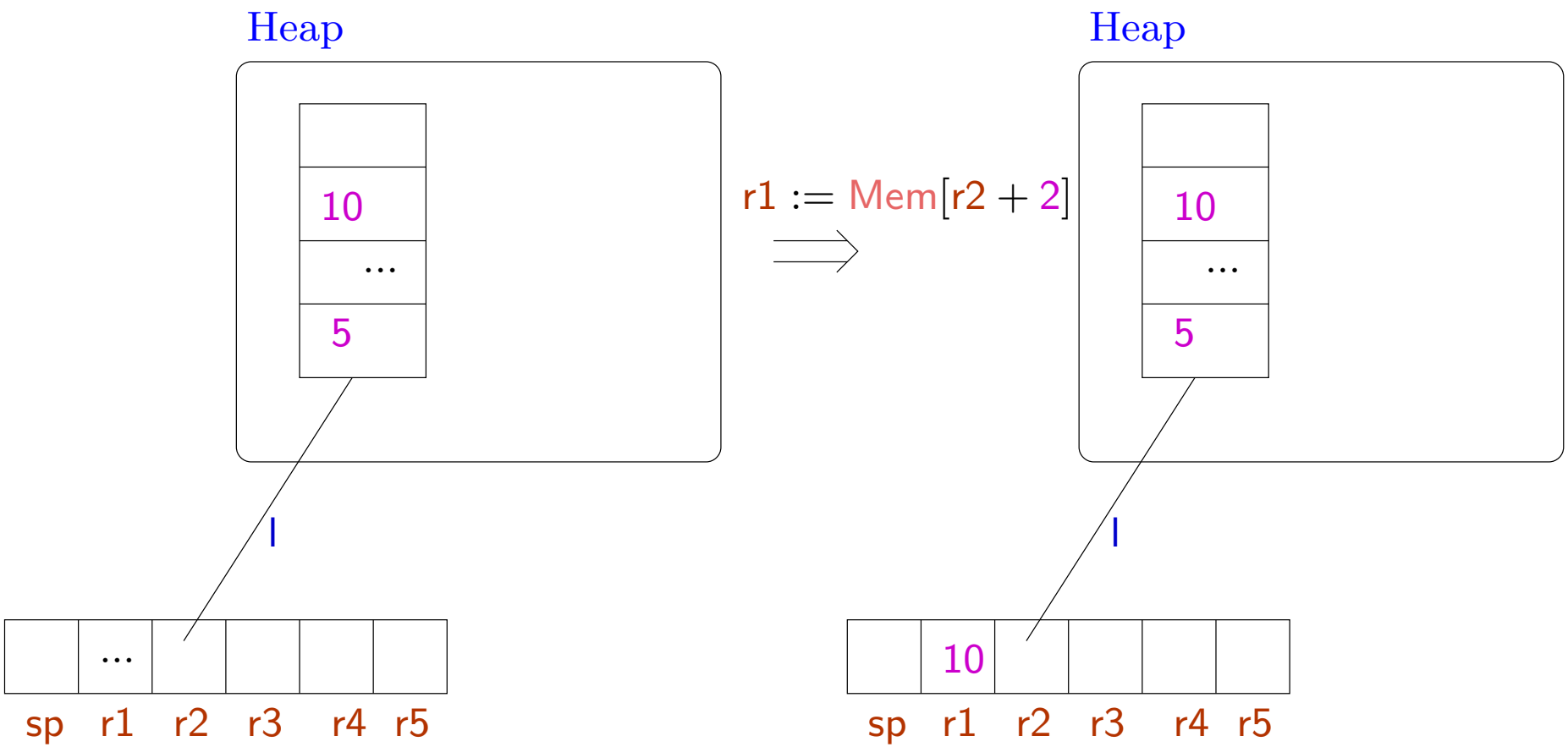
Loading shared data

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

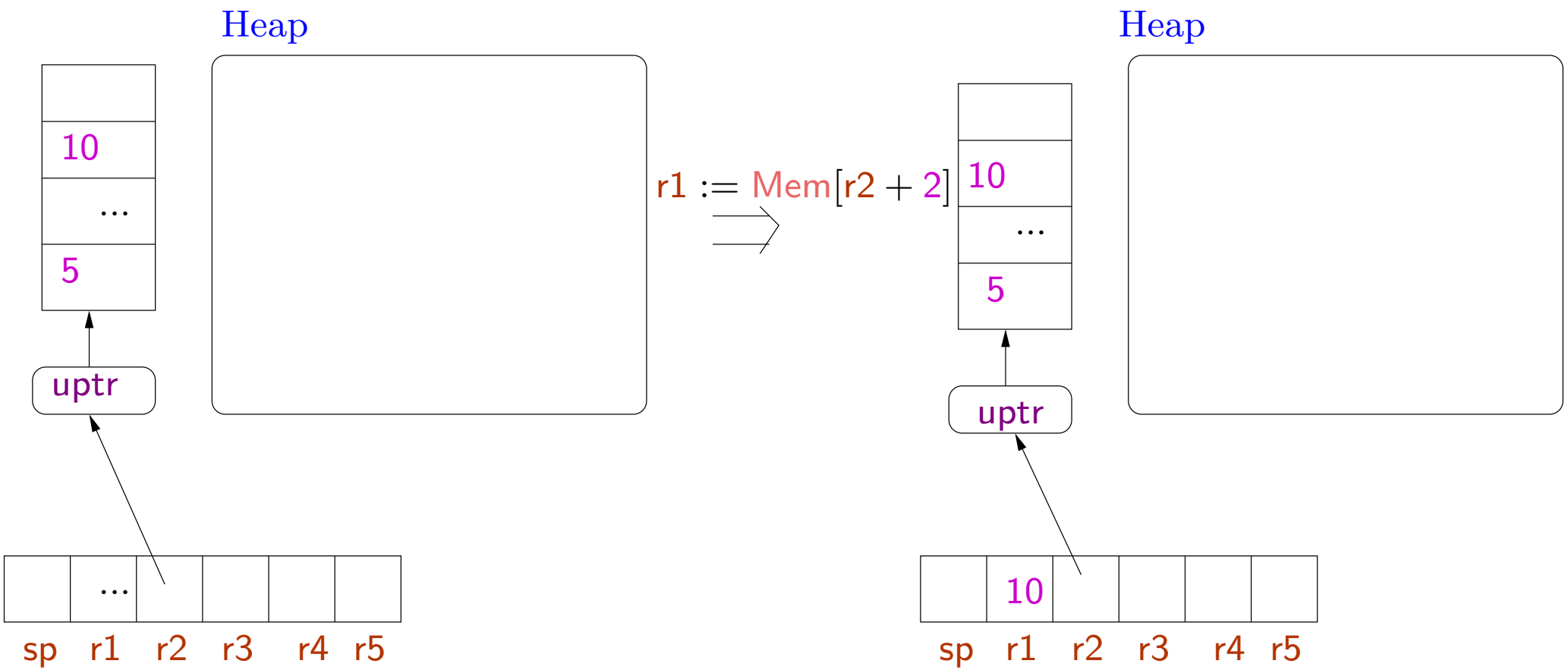
Loading unique data

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

Loading shared data



Loading unique data



Storing shared data

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

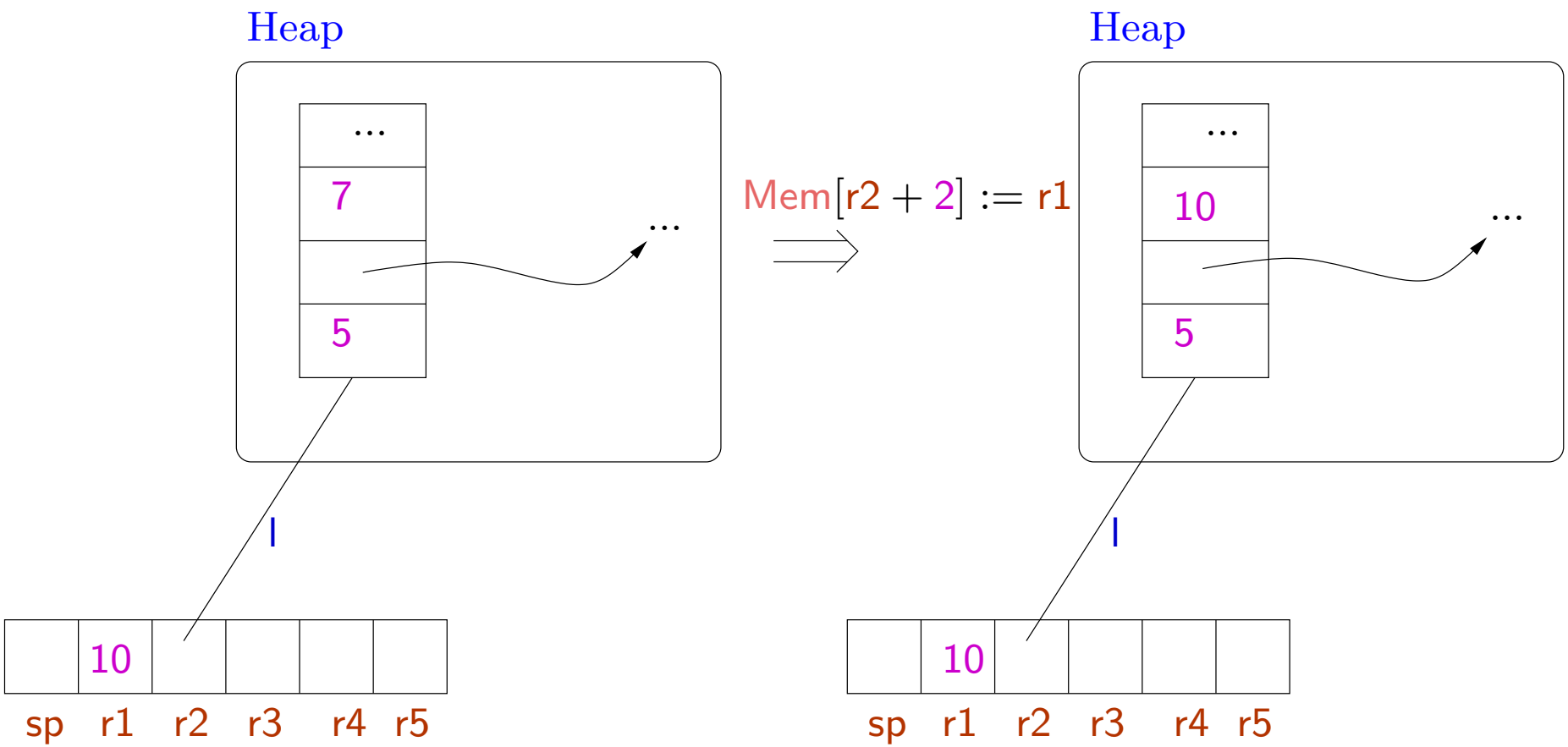
Storing shared data

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

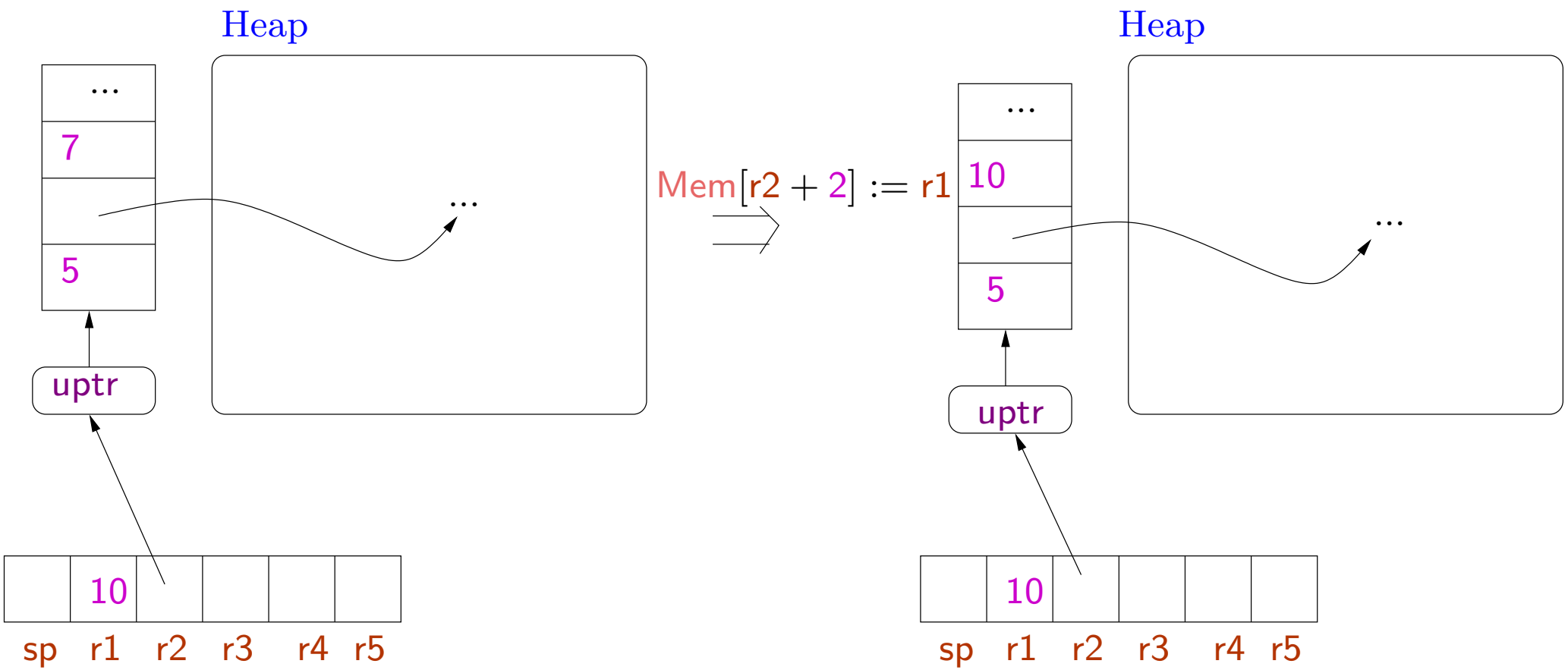
Storing unique data

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

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

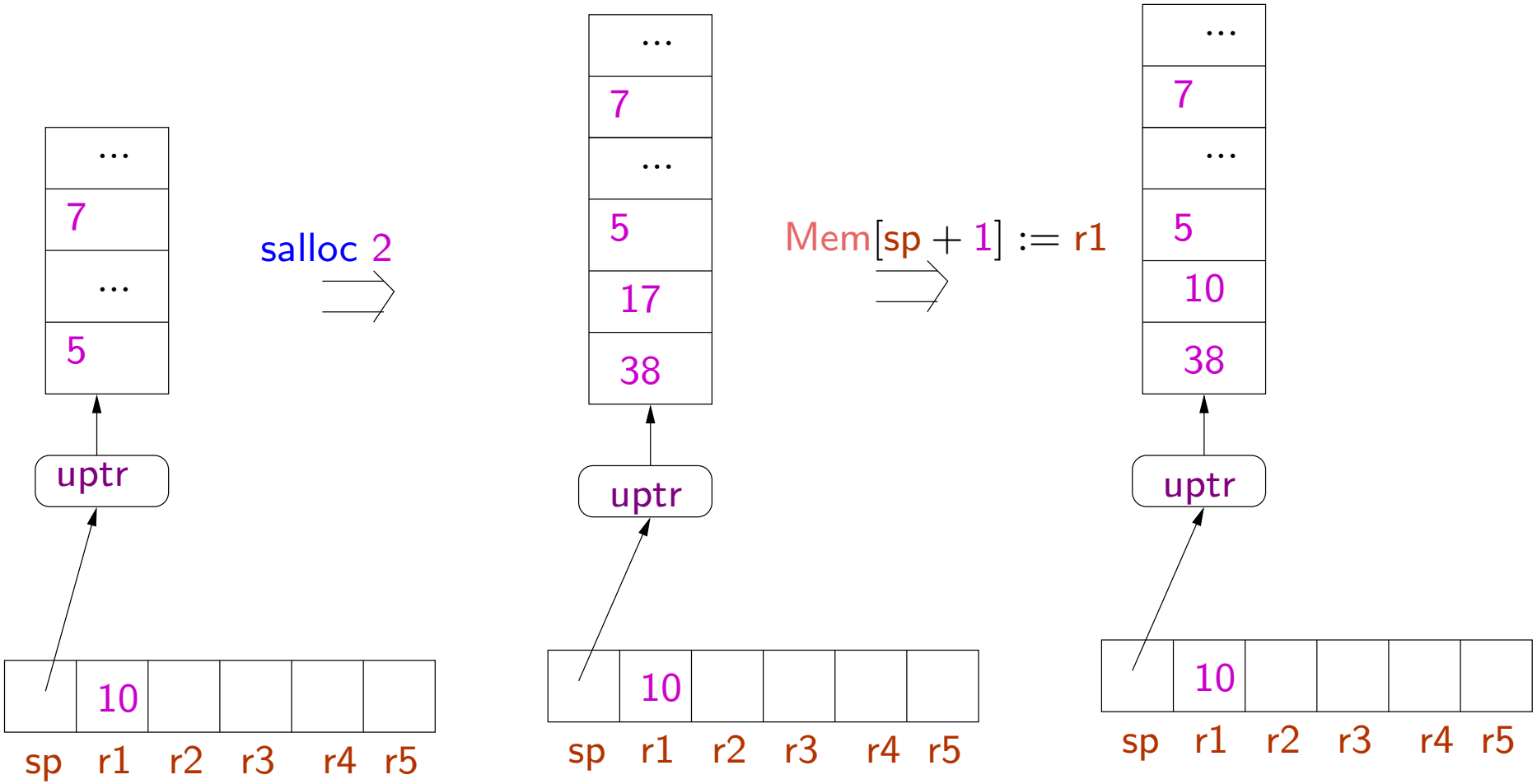
$$\frac{R(\mathsf{sp}) = \mathsf{uptr}\langle \nu_0, \ldots, \nu_p \rangle \quad p + n \leq \mathsf{MaxStack}}{(H, R, \mathsf{salloc}\ n; I) \longrightarrow (H, R \oplus \{\mathsf{sp} \mapsto \mathsf{uptr}\langle m_1, \ldots, m_n, \nu_0, \ldots, \nu_p \rangle\}, I)} \text{(E-Salloc)}$$

- 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.
- We have chosen the stack to grow upward: positive indexing as for other data tuples.

Deallocating space from the stack

$$\frac{R(\mathsf{sp}) = \mathsf{uptr}\langle \nu'_1, \ldots, \nu'_n, \nu_0, \ldots, \nu_p \rangle}{(H, R, \mathsf{sfree}\ n; I) \longrightarrow (H, R \oplus \{\mathsf{sp} \mapsto \mathsf{uptr}\langle \nu_0, \ldots, \nu_p \rangle\}, I)}\ \text{(E-Sfree)}$$

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


...
7
...
5
salloc 2
uptr
10
sp r1 r2 r3 r4 r5
...
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-0

TAL-0 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 r1 := malloc 3 makes the register r1 to be of type $\mathsf{uptr}\langle\mathsf{Int},\mathsf{Int},\mathsf{Int}\rangle$.

The instruction commit r2 transforms the type of register 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 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 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.

The TAL-1 type system

$\tau ::=$		operand types
	$\mathsf{Int} \mid \mathsf{Code}(\Gamma)$	
	$\mid \mathsf{ptr}(\sigma)$	shared pointer types
	$\mid \mathsf{uptr}(\sigma)$	unique pointer types
	$\mid \forall\rho \cdot \tau$	quantification over allocated types
$\sigma ::=$		allocated types
	ϵ	empty tuple type
	τ	one operand
	$\langle\sigma_1, \sigma_2\rangle$	pair
	ρ	allocated type variable

operand types are for operands and allocated data types are for tuples.

As before register file types Γ are of the form $\{\mathsf{sp} : \tau, \mathsf{r1} : \tau_1, \ldots, \mathsf{rk} : \tau_k\}$ where τ, τ_i are operand types.

Similarly heap types Ψ map labels to operand types.

We consider

$$\langle\langle\sigma_1, \sigma_2\rangle, \sigma_3\rangle = \langle\sigma_1, \langle\sigma_2, \sigma_3\rangle\rangle = \langle\sigma_1, \sigma_2, \sigma_3\rangle$$
$$\langle\sigma, \epsilon\rangle = \langle\epsilon, \sigma\rangle = \sigma$$
$$\ldots$$

Typing rules

Typing rules

Tuples

$$\frac{\forall 1 \leq i \leq n \cdot \Psi, \Gamma \vdash \nu_i : \tau_i}{\Psi, \Gamma \vdash \langle \nu_1, \ldots, \nu_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle} \text{ (T-Tuple)}$$

Typing rules

Tuples

$$\frac{\forall 1 \leq i \leq n \cdot \Psi, \Gamma \vdash \nu_i : \tau_i}{\Psi, \Gamma \vdash \langle \nu_1, \ldots, \nu_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle} \text{(T-Tuple)}$$

$$\frac{\Psi, \Gamma \vdash h : \sigma}{\Psi, \Gamma \vdash \mathsf{uptr}(h) : \mathsf{uptr}(\sigma)} \text{(T-Uptr)}$$

Typing of instructions

The older rules of TAL-0 remain unmodified, except for the Mov instruction, where now copying of unique pointers should be prevented. Hence we have the following new rule.

$$\frac{\Psi, \Gamma \vdash \nu : \tau \quad \tau \neq \mathsf{uptr}(\sigma)}{\Psi \vdash r_d := \nu : \Gamma \to \Gamma \oplus \{r_d : \tau\}} \text{ (T-Mov1)}$$

Typing of instructions

The older rules of TAL-0 remain unmodified, except for the Mov instruction, where now copying of unique pointers should be prevented. Hence we have the following new rule.

$$\frac{\Psi, \Gamma \vdash \nu : \tau \quad \tau \neq \mathsf{uptr}(\sigma)}{\Psi \vdash r_d := \nu : \Gamma \to \Gamma \oplus \{r_d : \tau\}} \text{ (T-Mov1)}$$

We add new typing rules for the new instructions.

$$\frac{\mathsf{n} \geq 0}{\Psi \vdash r_d := \mathsf{malloc}\ \mathsf{n} : \Gamma \to \Gamma \oplus \{r_d : \mathsf{uptr}\langle \underbrace{\mathsf{Int}, \ldots, \mathsf{Int}}_{\mathsf{n}\ \text{times}} \rangle\}} \text{ (T-Malloc)}$$

malloc creates a unique pointer type.

$$\frac{\Psi, \Gamma \vdash r_d : \mathsf{uptr}(\sigma) \quad r_d \neq \mathsf{sp}}{\Psi \vdash \mathsf{commit}\ r_d : \Gamma \to \Gamma \oplus \{r_d : \mathsf{ptr}(\sigma)\}}\ (\text{T-Commit})$$

commit creates a shared pointer type.

r_d stores a (label) pointer to the value which has now been moved into the heap.

$$\frac{\Psi, \Gamma \vdash r_s : \mathsf{ptr}\langle \tau_0, \ldots, \tau_n, \sigma \rangle}{\Psi \vdash r_d := \mathsf{Mem}[r_s + \mathsf{n}] : \Gamma \to \Gamma \oplus \{r_d : \tau_n\}} \text{ (T-Ld-S)}$$

$$\frac{\Psi, \Gamma \vdash r_s : \mathsf{ptr}\langle \tau_0, \ldots, \tau_n, \sigma \rangle}{\Psi \vdash r_d := \mathsf{Mem}[r_s + \mathsf{n}] : \Gamma \to \Gamma \oplus \{r_d : \tau_n\}} \text{ (T-Ld-S)}$$

$$\frac{\Psi, \Gamma \vdash r_s : \mathsf{uptr}\langle \tau_0, \ldots, \tau_n, \sigma \rangle}{\Psi \vdash r_d := \mathsf{Mem}[r_s + \mathsf{n}] : \Gamma \to \Gamma \oplus \{r_d : \tau_n\}} \text{ (T-Ld-U)}$$

$$\frac{\Psi, \Gamma \vdash r_d : \mathsf{ptr}\langle \tau_0, \ldots, \tau_n, \sigma \rangle \quad \Psi, \Gamma \vdash r_s : \tau_n \quad \tau_n \neq \mathsf{uptr}(\sigma')}{\Psi \vdash \mathsf{Mem}[r_d + \mathsf{n}] := r_s : \Gamma \to \Gamma} \text{ (T-St-S)}$$

Updating shared data should not involve a change in type.

$$\frac{\Psi, \Gamma \vdash r_d : \mathsf{ptr}\langle \tau_0, \ldots, \tau_n, \sigma \rangle \quad \Psi, \Gamma \vdash r_s : \tau_n \quad \tau_n \neq \mathsf{uptr}(\sigma')}{\Psi \vdash \mathsf{Mem}[r_d + \mathsf{n}] := r_s : \Gamma \to \Gamma} \text{ (T-St-S)}$$

Updating shared data should not involve a change in type.

$$\frac{\Psi, \Gamma \vdash r_d : \mathsf{uptr}\langle \tau_0, \ldots, \tau_n, \sigma \rangle \quad \Psi, \Gamma \vdash r_s : \tau \quad \tau \neq \mathsf{uptr}(\sigma')}{\Psi \vdash \mathsf{Mem}[r_d + \mathsf{n}] := r_s : \Gamma \to \Gamma \oplus \{r_d : \mathsf{uptr}\langle \tau_0, \ldots, \tau_{n-1}, \tau, \sigma \rangle\}} \text{ (T-St-U)}$$

$$\frac{\Psi, \Gamma \vdash \mathsf{sp} : \mathsf{uptr}(\sigma) \quad \mathsf{n} \geq 0}{\Psi \vdash \mathsf{salloc}\ \mathsf{n} : \Gamma \to \Gamma \oplus \{\mathsf{sp} : \mathsf{uptr}\langle \underbrace{\mathsf{Int}, \ldots, \mathsf{Int}}_{\mathsf{n}\ \text{times}}, \sigma \rangle\}} \ (\Gamma\text{-Salloc})$$

$$\frac{\Psi, \Gamma \vdash \mathsf{sp} : \mathsf{uptr}(\sigma) \quad \mathsf{n} \geq 0}{\Psi \vdash \mathsf{salloc}\ \mathsf{n} : \Gamma \to \Gamma \oplus \{\mathsf{sp} : \mathsf{uptr}\langle \underbrace{\mathsf{Int}, \ldots, \mathsf{Int}}_{\mathsf{n}\ \text{times}}, \sigma \rangle\}} \text{ (T-Salloc)}$$

$$\frac{\Psi, \Gamma \vdash \mathsf{sp} : \mathsf{uptr}\langle \tau_1, \ldots, \tau_n, \sigma \rangle}{\Psi \vdash \mathsf{sfree}\ \mathsf{n} : \Gamma \to \Gamma \oplus \{\mathsf{sp} : \mathsf{uptr}(\sigma)\}} \text{ (T-Sfree)}$$

$$\frac{\Psi, \Gamma \vdash \mathsf{sp} : \mathsf{uptr}(\sigma) \quad \mathsf{n} \geq 0}{\Psi \vdash \mathsf{salloc}\ \mathsf{n} : \Gamma \to \Gamma \oplus \{\mathsf{sp} : \mathsf{uptr}\langle \underbrace{\mathsf{Int}, \ldots, \mathsf{Int}}_{\mathsf{n}\ \text{times}}, \sigma \rangle\}} \text{ (T-Salloc)}$$

$$\frac{\Psi, \Gamma \vdash \mathsf{sp} : \mathsf{uptr}\langle \tau_1, \ldots, \tau_n, \sigma \rangle}{\Psi \vdash \mathsf{sfree}\ \mathsf{n} : \Gamma \to \Gamma \oplus \{\mathsf{sp} : \mathsf{uptr}(\sigma)\}} \text{ (T-Sfree)}$$

Stack underflows are ruled out by the type system.

What about stack overflows??

The type system is not powerful enough to keep track of the size of stack.

Hence Code leading to stack overflow will be well-typed, violating safety.

To ensure type safety, we add new evaluation rules in case of stack overflow.

The type system is not powerful enough to keep track of the size of stack.

Hence Code leading to stack overflow will be well-typed, violating safety.

To ensure type safety, we add new evaluation rules in case of stack overflow.

$$\frac{R(\mathsf{sp}) = \mathsf{uptr}\langle \nu_0, \ldots, \nu_p \rangle \quad p + n > \mathsf{MaxStack}}{(H, R, \mathsf{salloc}\ n; I) \rightarrow \mathsf{StackOverflow}}\ (\text{E-Overflow1})$$

Where StackOverflow is a new special machine state.

This is similar to "error" terms in our previous discussion on type safety.

The rules for typing instruction sequences, register files, heaps and machine states are as for TAL-0.

We further require rules for quantifying over allocated type variables, and for generating instances.

The rules for typing instruction sequences, register files, heaps and machine states are as for TAL-0.

We further require rules for quantifying over allocated type variables, and for generating instances.

$$\frac{\Psi \vdash I : \tau}{\Psi \vdash I : \forall \rho \cdot \tau} \text{ (T-Gen)}$$

ρ is an allocated type variable possibly occurring in τ.

Type of labels can be instantiated by the following rule.

We replace occurrences of ρ by any desired type τ'.

$$\frac{\Psi, \Gamma \vdash \nu : \forall \rho \cdot \tau}{\Psi, \Gamma \vdash \nu : \tau[\rho \mapsto \tau']} \text{ (T-Inst)}$$

Example

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

We would like to assign to this instruction sequence, the type

$\tau = \forall \mathsf{s} \cdot \mathsf{Code}\{\Gamma\}$ where

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

where allocated type variable sp represents an arbitrary chunk of memory.

Let $\Gamma_1 = \Gamma \oplus \{\mathsf{r1} : \mathsf{Int}\}$ and $\Gamma_2 = \Gamma_1 \oplus \{\mathsf{sp} : \mathsf{uptr}(\mathsf{s})\}$.

For any heap type Ψ we have the following typing derivation.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Psi, \Gamma_2 \vdash \mathsf{r3} : \mathsf{Code}\{\mathsf{sp} : \mathsf{uptr}(\mathsf{s}), \mathsf{r1} : \mathsf{Int}, \mathsf{r2}, \mathsf{r3} : \mathsf{Top}\} \qquad \mathsf{Code}(\Gamma_2) \sqsubseteq \mathsf{Code}\{\ldots\}}\ (\text{T-Sub})}{\Psi, \Gamma_2 \vdash \mathsf{r3} : \mathsf{Code}(\Gamma_2)}\ (\text{T-Jump})}{\Psi \vdash \ \mathsf{jump}\ \mathsf{r3} : \mathsf{Code}(\Gamma_2)}$$

$$
\cfrac{\cfrac{\begin{array}{c}\vdots\\ \Psi, \Gamma_2 \vdash \mathsf{r3} : \mathsf{Code}\{\mathsf{sp} : \mathsf{uptr}(\mathsf{s}), \mathsf{r1} : \mathsf{Int}, \mathsf{r2}, \mathsf{r3} : \mathsf{Top}\}\end{array} \qquad \mathsf{Code}(\Gamma_2) \sqsubseteq \mathsf{Code}\{\ldots\}}{\Psi, \Gamma_2 \vdash \mathsf{r3} : \mathsf{Code}(\Gamma_2)}\ (\text{T-Sub})}{\Psi \vdash \ \mathsf{jump}\ \mathsf{r3} : \mathsf{Code}(\Gamma_2)}\ (\text{T-Jump})
$$

$$
\cfrac{\cfrac{\Psi, \Gamma_1 \vdash \mathsf{sp} : \mathsf{uptr}\langle \mathsf{Int}, \mathsf{s}\rangle}{\Psi \vdash \mathsf{sfree}\ 1 : \Gamma_1 \to \Gamma_2}\ (\text{T-Sfree}) \qquad \begin{array}{c}\vdots\\ \Psi \vdash \ \mathsf{jump}\ \mathsf{r3} : \mathsf{Code}(\Gamma_2)\end{array}}{\Psi \vdash \mathsf{sfree}\ 1;\ \mathsf{jump}\ \mathsf{r3} : \mathsf{Code}(\Gamma_1)}\ (\text{T-Seq})
$$

$$
\cfrac{\cfrac{\begin{array}{c}\vdots\\ \Psi \vdash \mathsf{r1} := 0 : \Gamma \to \Gamma_1\end{array} \qquad \begin{array}{c}\vdots\\ \Psi \vdash \mathsf{sfree}\ 1;\ \mathsf{jump}\ \mathsf{r3} : \mathsf{Code}(\Gamma_1)\end{array}}{\Psi \vdash \mathsf{r1} := 0; \mathsf{sfree}\ 1;\ \mathsf{jump}\ \mathsf{r3} : \mathsf{Code}(\Gamma)}\ (\text{T-Seq})}{\Psi \vdash \mathsf{r1} := 0; \mathsf{sfree}\ 1;\ \mathsf{jump}\ \mathsf{r3} : \forall \mathsf{s} \cdot \mathsf{Code}(\Gamma)}\ (\text{T-Gen})
$$