Language Based Security

Kumar Neeraj Verma

TU München

Summer Semester 2006

Organization

Lectures: Wednesday, 10:15 - 11:45

Tutorials: Friday, 10:15 - 11:00

Starting 12.05.06

Schein: Written examination

Planned contents

- Buffer overflow attacks
  $\longrightarrow$ Prevention using program analysis
- Security issues in Java
- Type systems for safety
- Bytecode verification and proof carrying code
- Techniques for access control and information flow analysis

Computer Security

Some goals

- Confidentiality of information
- Authenticity
- Preventing other improper behavior like not paying for services
- Ensuring availability of services
- Preventing damage of information

Challenges

- Increasing complexity of software; frequent updates
- Untrusted programs
- Computer systems are not isolated
- Numerous possibilities for attacks: webpages with executables, emails, cookies, . . .
- Financial cost of an insecurity could be huge

The Morris Worm, 1988

- One of the first known internet worms.
- Among others it exploited a buffer overflow vulnerability in fingerd.
- A worm at an infected host copied itself to other hosts by exploiting vulnerabilities. The number of copies running at a host slowed it down to the point of being unusable.
- An estimated 6000 machines (10 % of hosts at that time) were infected.
- Huge financial losses were incurred because infected hosts were unable to continue functioning.

New buffer overflow vulnerabilities still continue to be found.

The MS-SQL Slammer worm, 2003

- Exploited a buffer overflow vulnerability in Micorsoft SQL server announced in 2002.
- Affected more than 75000 hosts, most of them within the first 10 minutes.

The Code Red worm, 2001

- Exploited a buffer overflow vulnerability in Microsoft's IIS web server.

Buffer overflows

- The C language allows access to arbitrary memory locations through improper use of pointers.
- This leads to a typical programming error of accessing a buffer (array) beyond the space allocated for it.
- Typically exploited by stack smashing attacks involving overflowing buffers on the stack to overwrite the return address.
- Data extracted from CERT advisories show that buffer overflows are responsible for nearly half of todays vulnerabilities.

Pointers and arrays in C

For any variable we can obtain the corresponding memory location using the `&` operator. The `*` operator gives the value stored at a memory location.

```
main() {
   int  x = 10;
   int  *p;
   printf(”x = %d\n”,x);
   p = &x;
   *p = 20;
   printf(”x = %d\n”, x);
}
```

Output:

```
x = 10
x = 20
```

This leads to pointer arithmetic:

```
main() {
   int  x,  y;
   x = 10;
   printf(”x = %d\n”,x);
   *((&y)+1) = 20;
   printf(”x = %d\n”,x);
}
```

Output:

```
x = 10
x = 20
```

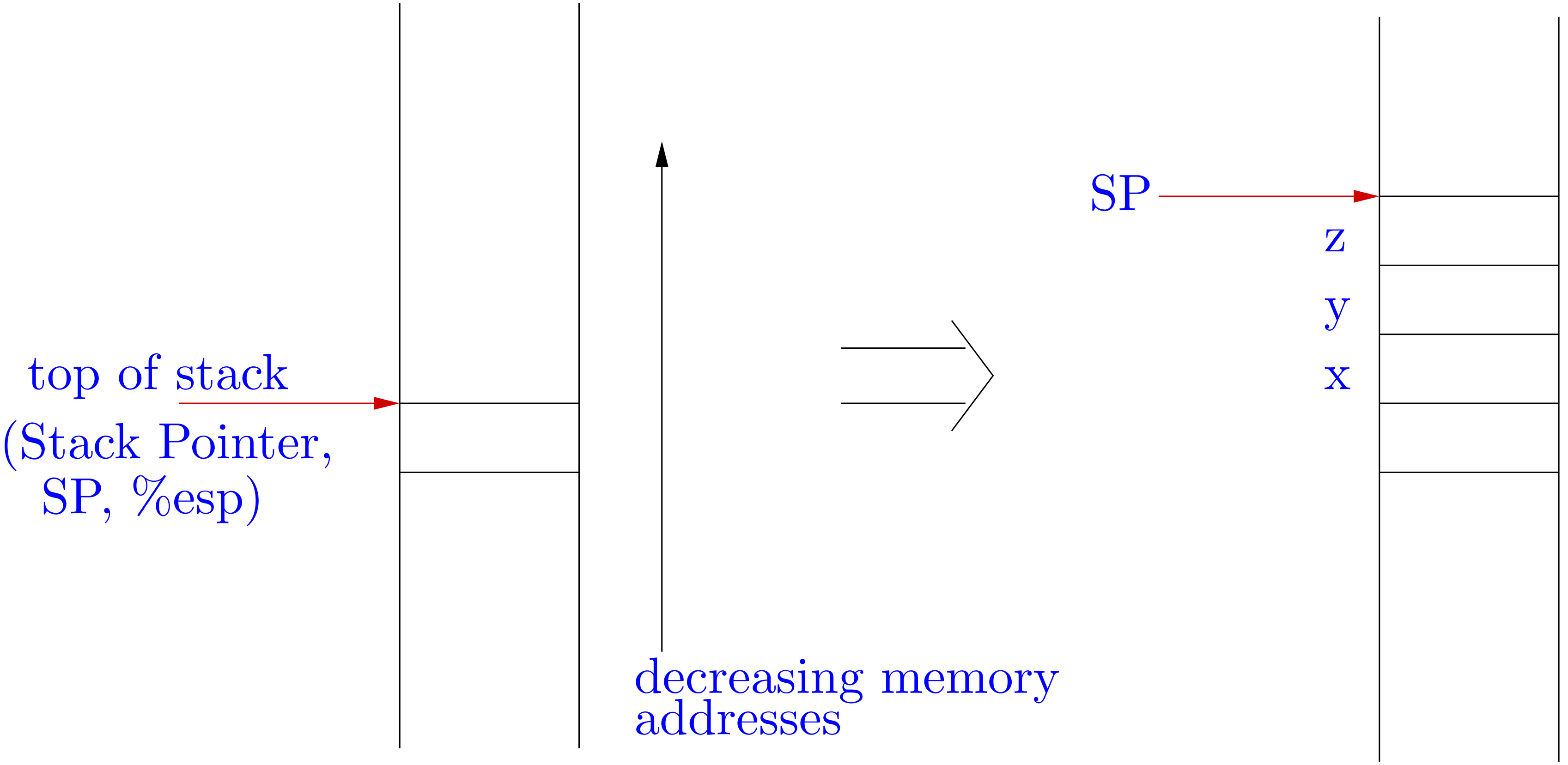
C allows access to arbitrary memory locations through pointers.

Here we need to know that `x` and `y` are allocated space on consecutive locations.

The declaration

```
int x,y,z;
```

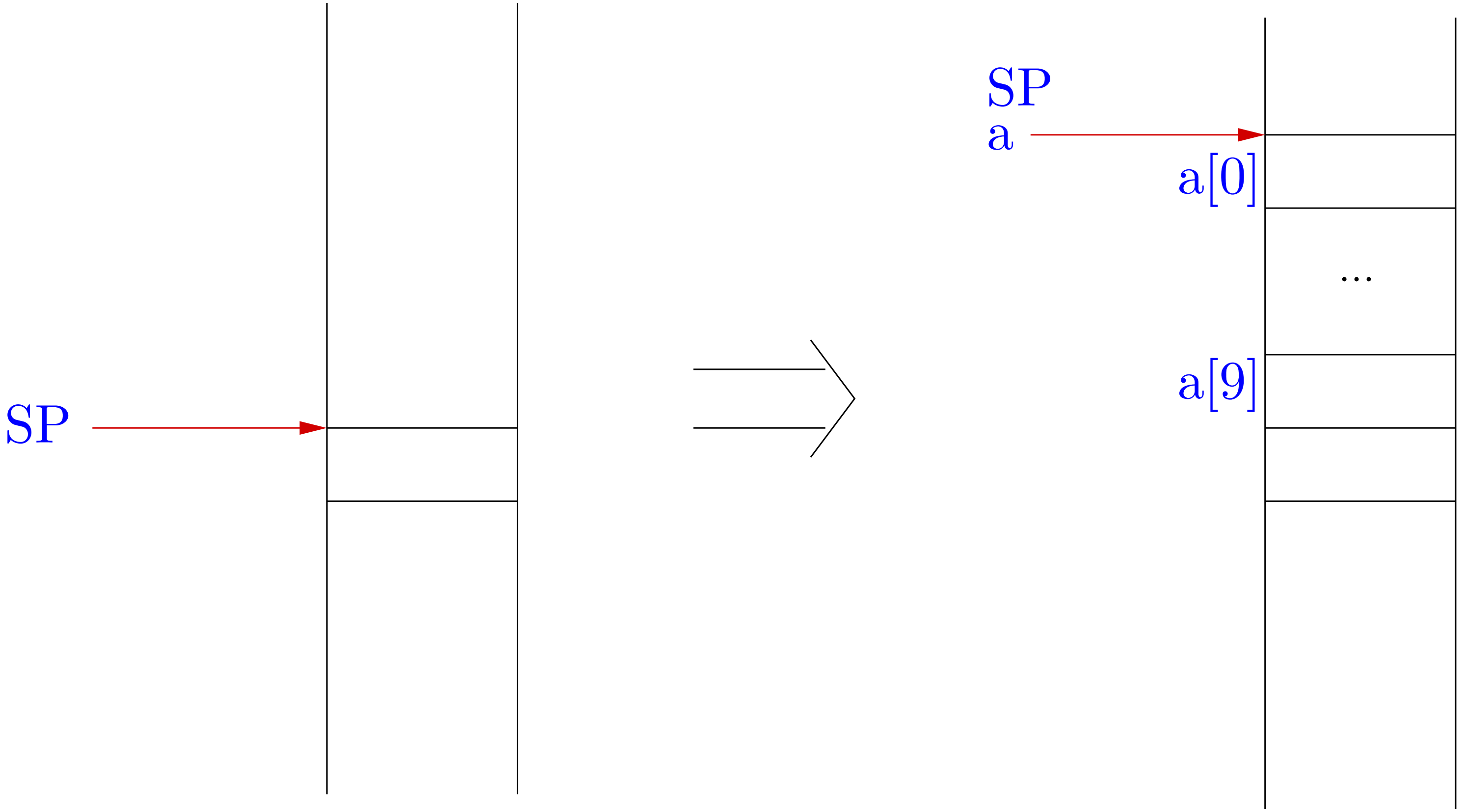
leads to allocation of space on the stack as follows.



Allocating space for arrays on the stack:

```
int a[10];
```

`a` is also the address where `a[0]` is stored. `a[5]=10` is same as `*(a+5)=10`.



Enough ingredients for errors introduced by careless programmers!

```
main() {
  int x,a[10], i;

  x = 10;
  printf(”x = %d\n”,x);
  for (i=0; i<=15; i++) a[i]=20;
  printf(”x = %d\n”, x);

  /* Code may require adjustment to
     machine and compiler */
}
```

```
x = 10
x = 20
```

Out of bound access in array `a`, leading to modification of value of `x`.

No checks enforced by the C language!

Compare with Java ⟶ a strongly typed language

```
public class Array1 {
  public static void main (String args[]) {
    int x, a[] = new int[10], i;

    x = 10;
    System.out.println (”x=” + x);
    for (i=0; i<=15; i++) a[i]=20;
    System.out.println (”x=” + x);
  }
}
```

```
x=10
Exception in thread ”main” java.lang.ArrayIndexOutOfBoundsException: 10
        at Array1.main(Array1.java:7)
```

Exceptions may then be caught and some other action taken.

```
public class Array2 {
  public static void main (String args []) {
    int x, a [] = new int [10], i ;

    x = 10;
    System.out.println (”x=” + x);
    for (i=0; i<=15; i++)
      try { a[i]=20; } catch (Exception e) { }
    System.out.println (”x=” + x);
  }
}
```

```
x=10
x=10
```

Function calls and stack frames

- Each time a function is called, space must be allocated for the local variables of the function. This region of the stack is called the stack frame for this function call.

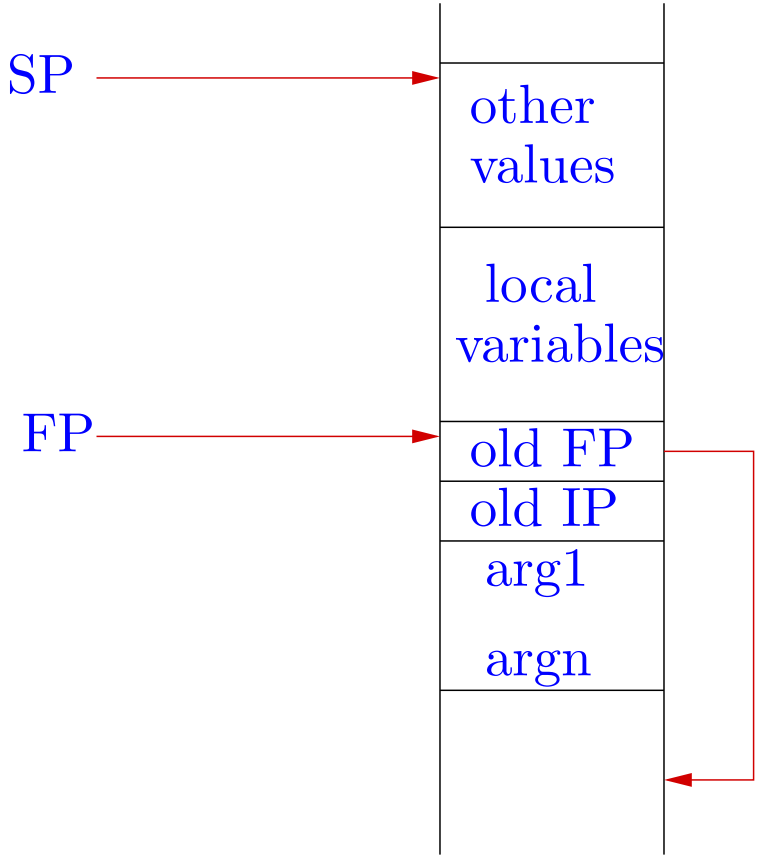
  ⇒ Use a Frame Pointer (FP, %ebp) to indicate the location of the current frame. This allows easy access to the local variables at runtime.

- On return from a function call, execution must continue from the next instruction after the function call.

  ⇒ Store the old instruction pointer (PC) in the stack frame.

- On return from a function, the current stack frame is popped out and execution continues with the previous stack frame.

  ⇒ Store the old FP on the stack.



A simple example of function call.

```
/* function.c */
void f (int x, int y) {
  int a,b,c;
}

int main () {
  f (10, 20);
}
```

Let's see the compiled code produced.

```
$ gdb function
```

. . .

The caller:

```
(gdb) disassemble main
 ...
0x804832f <main+19>: push   $0x14
0x8048331 <main+21>: push   $0xa
0x8048333 <main+23>: call   0x8048314 <f>
 ...
```

The arguments are pushed on to the stack and the function is called.

The caller:

```
(gdb) disassemble main
 ...
0x804832f <main+19>: push   $0x14
0x8048331 <main+21>: push   $0xa
0x8048333 <main+23>: call   0x8048314 <f>
 ...
```

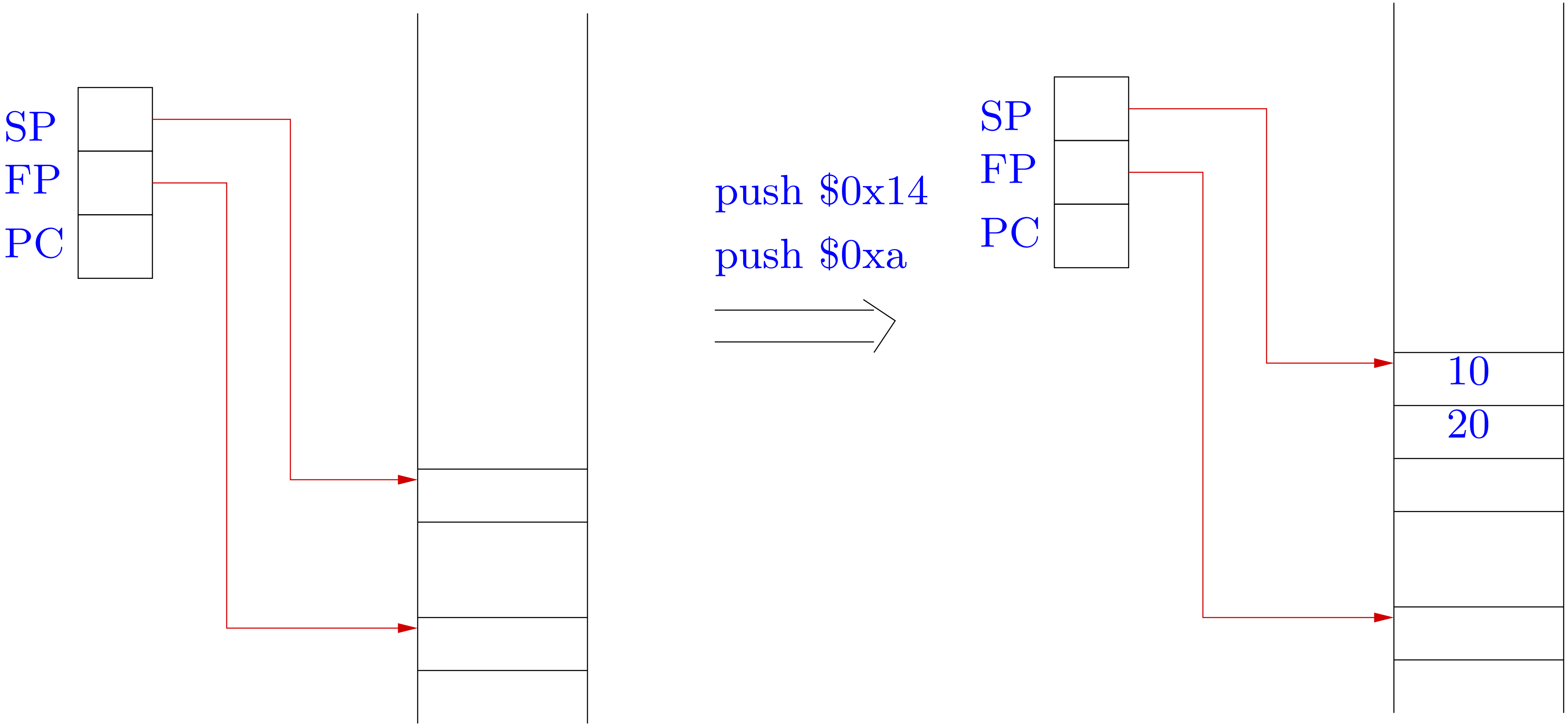
The arguments are pushed on to the stack and the function is called.

And the callee…

```
0x8048314 <f>:      push   %ebp
0x8048315 <f+1>:    mov    %esp,%ebp
0x8048317 <f+3>:    sub    $0xc,%esp
0x804831a <f+6>:    leave
0x804831b <f+7>:    ret
```

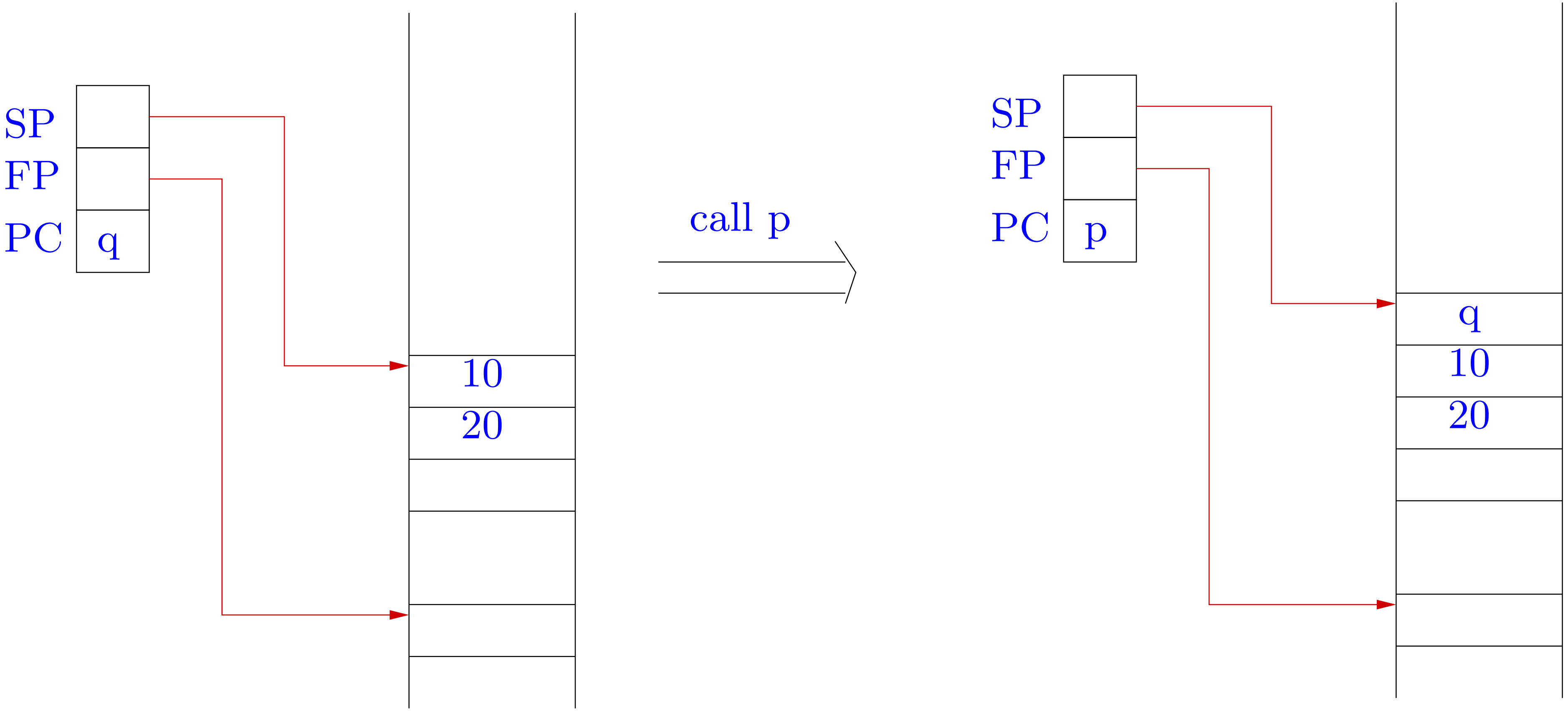
- Save old FP, update FP
- Allocate space for local variables, do computations
- Restore FP, pop saved FP from stack
- Return (restore PC, pop saved PC from stack)

At run time: pushing arguments

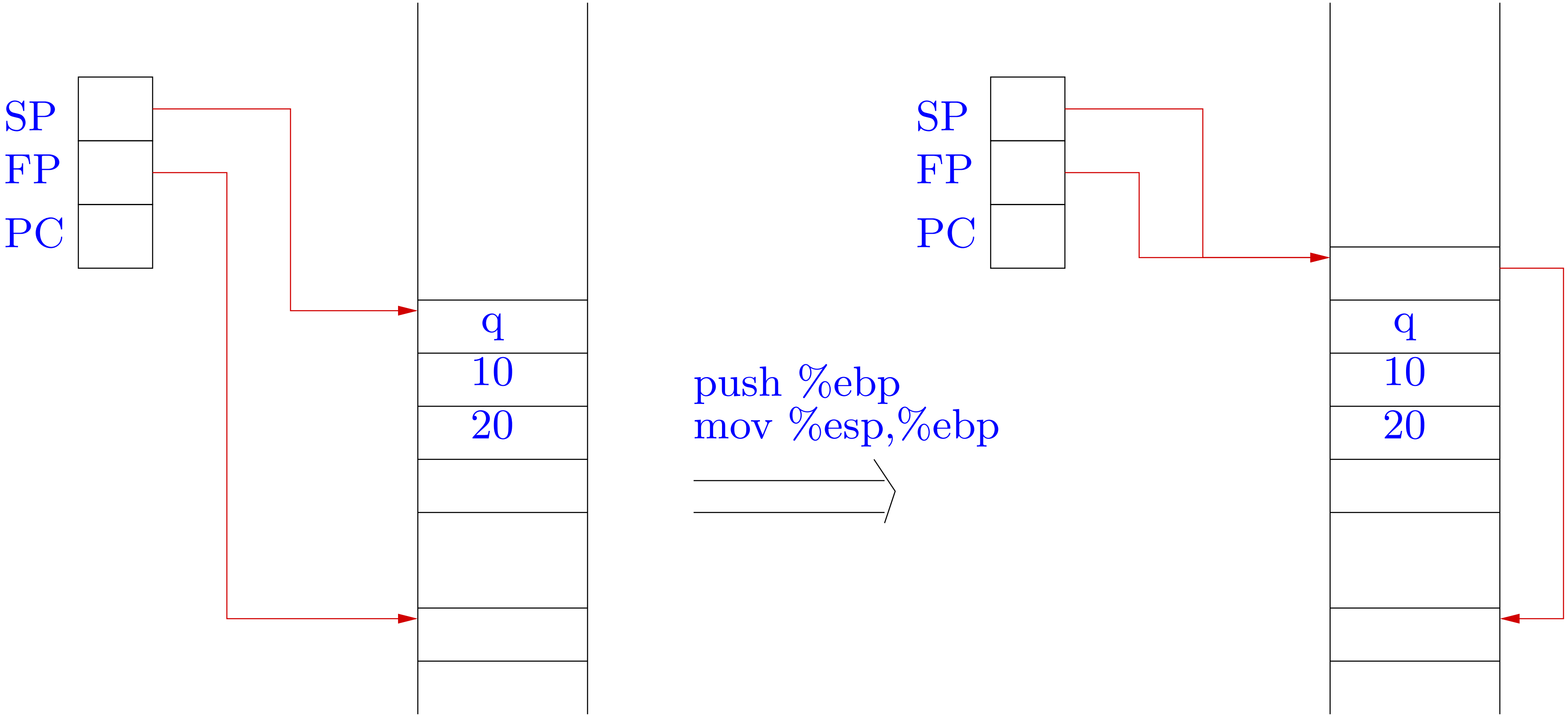


```
push $0x14
push $0xa
```

Calling function: saving PC and updating PC

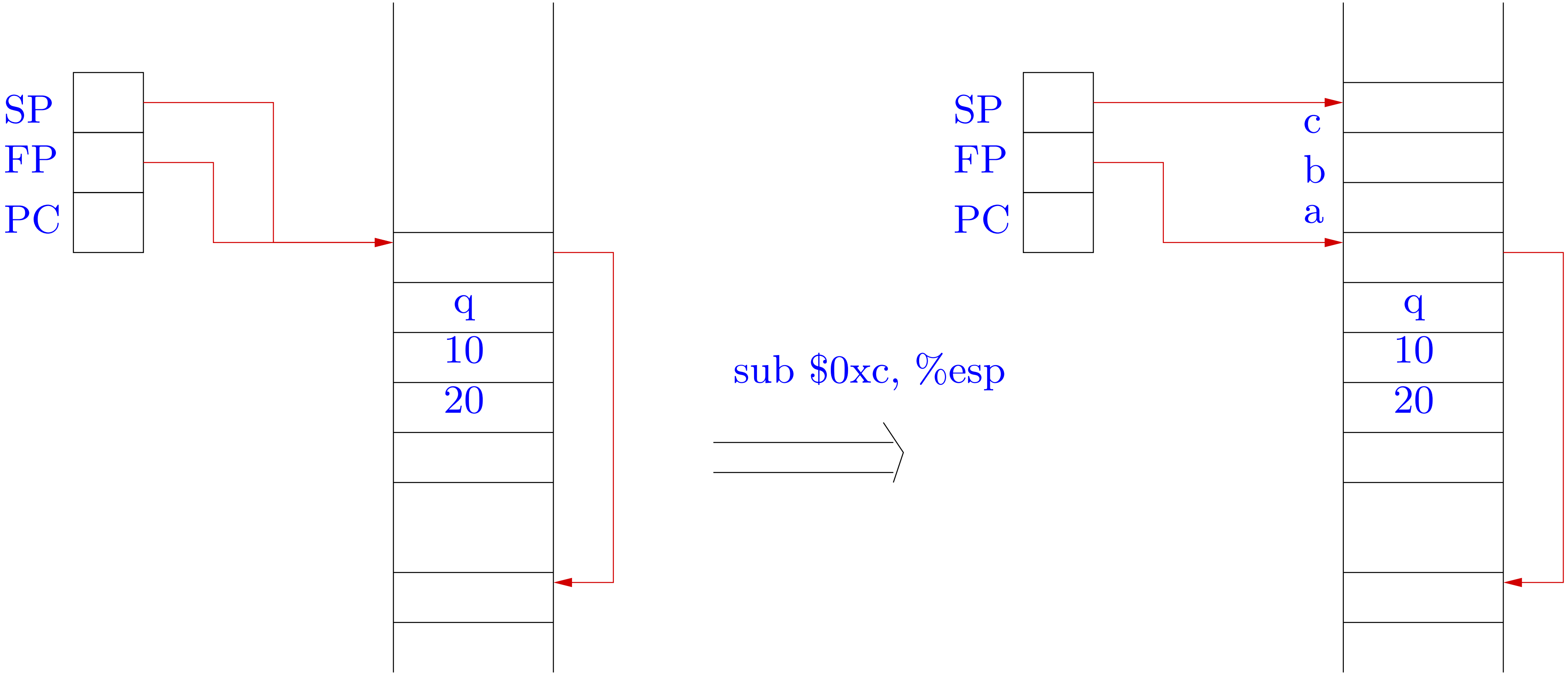


Inside callee: saving FP and updating FP

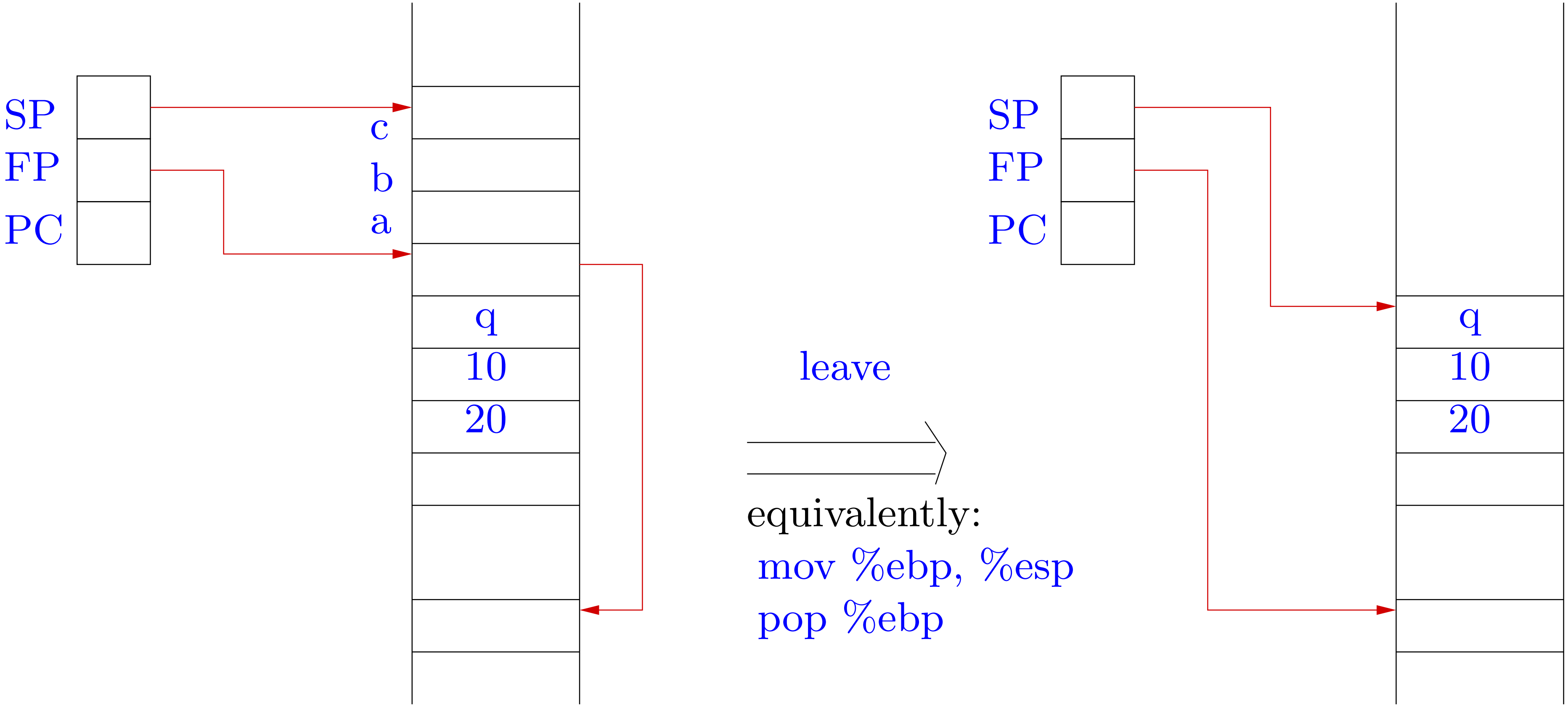


push %ebp
mov %esp,%ebp

Allocating space for local variables



End of callee: restoring FP and popping saved FP

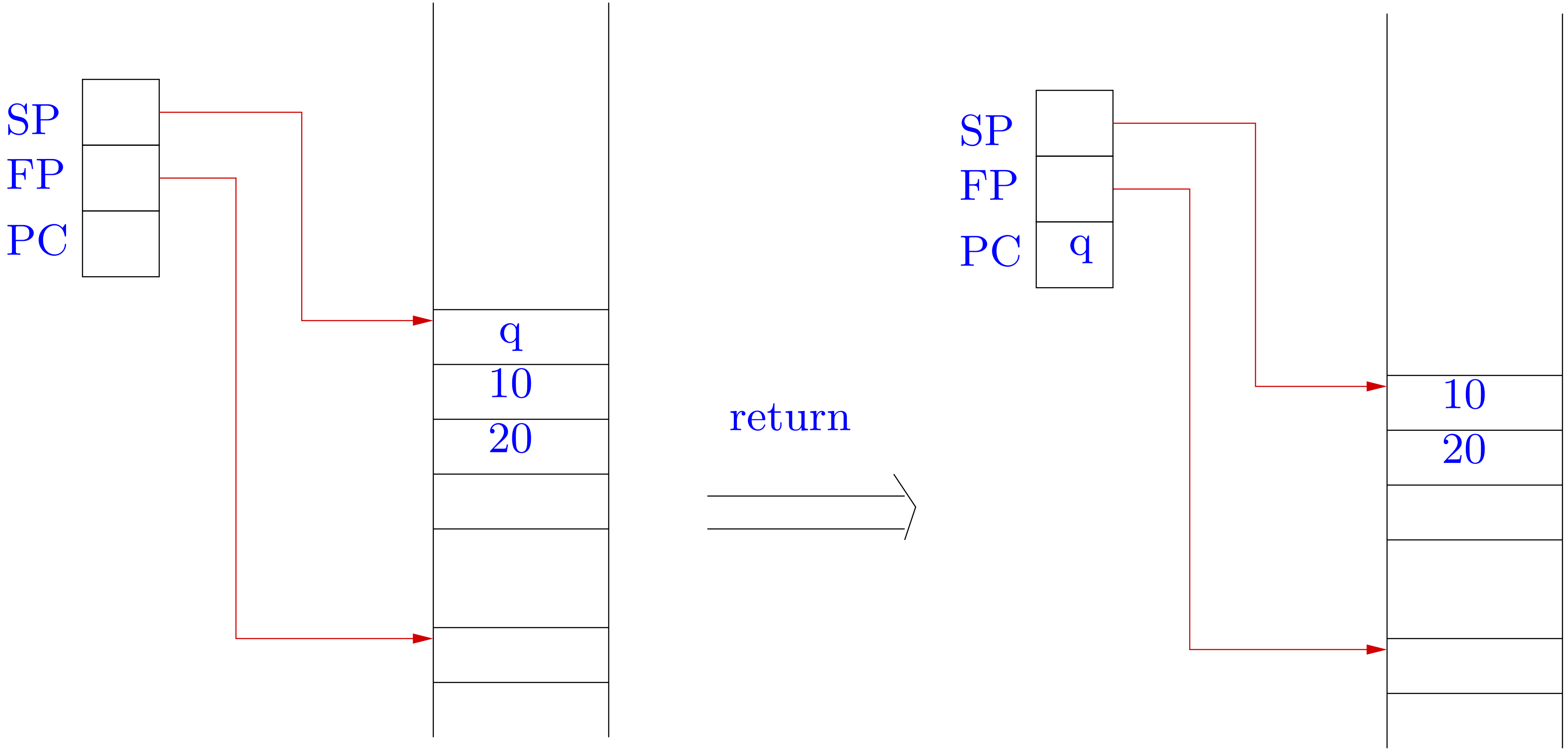


leave

equivalently:

```
mov %ebp, %esp
pop %ebp
```

Returning: restoring PC and popping saved PC



The return address is stored on the stack.

⇒ it can also be overwritten to point to arbitrary code!!!

```
void f () {
int a[10];
a[15] += 7;
}
```

```
main () {
  int x = 10;
  f ();
  x = 20;
  printf (”x=%d!\n”,x);
}
```

Output:

```
x=10!
```

We have skipped the instruction $x = 20;$!

- Where is the return address stored (a[15])?
- What should be the new return address (increment by 7)?

Organization of the stack: a[0], . . . , a[9], old FP, old PC

Hence the return address is at the location a[11].

Organization of the stack: a[0], . . . , a[9], old FP, old PC

Hence the return address is at the location a[11].

Not always!! Compiler optimizations may create blank spaces between array a and the following data.

⇒ Look at the compiled code.

Organization of the stack: a[0], . . . , a[9], old FP, old PC

Hence the return address is at the location a[11].

Not always!! Compiler optimizations may create blank spaces between array a and the following data.

⇒ Look at the compiled code.

```
0x8048344 <f>:        push   %ebp
0x8048345 <f+1>:      mov    %esp,%ebp
0x8048347 <f+3>:      sub    $0x38,%esp
 ...
```

Space allocated after old FP is 0x38 = 56 = 4*14 bytes.

Hence return address is at address a[15]

```
...
0x8048369 <main+23>: call   0x8048344 <f>
0x804836e <main+28>: movl   $0x14,0 xfffffffc (%ebp)
0x8048375 <main+35>: sub    $0x8,%esp
...
```

Instruction x = 20; requires 35 - 28 = 7 bytes.

Hence we put a[15] +=7 in the function f in order to skip execution of this instruction.

⇒ Besides modifying data, we may cause arbitrary code to be executed!

Weaknesses can be exploited by users by supplying appropriate inputs.

```
int main (int argc, char *argv[]) {
  char s[1024];
  strcpy(s,argv[1]);
   ...
}
```

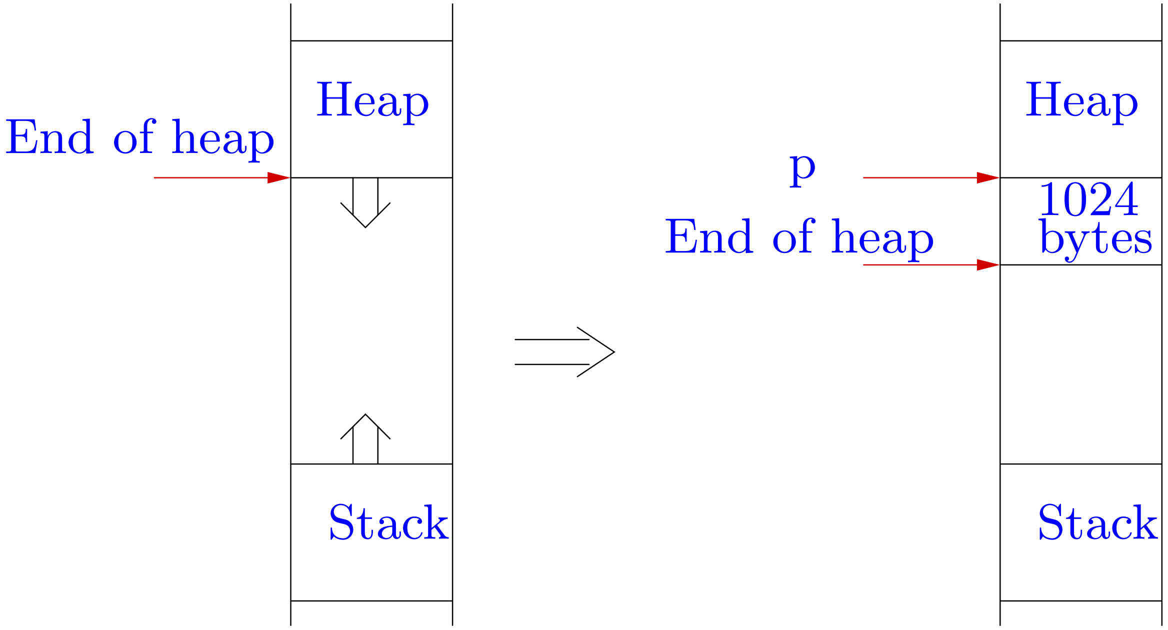
- An appropriate input is given to overwrite the return address,

- At the minimum, the program may abort abruptly.

- An ingenious attacker may get some desired code to be executed (shellcode) by providing it as a part of the input string!

Heap based overflows: buffer overflows in the heap instead of the stack.

```
char *p = (char *) malloc (1024);
```



Instead of overwriting return addresses, an attacker may overwrite important variables.

Further errors arise because of improper use of string library functions.

In C, the end of a string is indicated by the null character.

The statement

```
strcpy (s,t);
```

will keep copying characters starting from t till a null character is found, irrespective of space allocated for s and t.

```
i = strlen (s);
```

tries to find the first null charachter beyond s.

Some techniques for preventing buffer overflow attacks.

- Careful programming: e.g. use strncpy instead of strcpy.

- Make the stack region non-executable: however some applications make use of an executable stack.

- Compiler tools: save the return address at a safe place (data region).

- Run time checks: use a preloaded library which provides safer versions of standard unsafe functions.

Detecting buffer overflow vulnerabilities

- Static program analysis: automated analysis of programs without running them.

- an exact analysis of buffer overflow vulnerabilities is theoretically impossible.

$\Longrightarrow$ do approximate analysis:

  - we fail to detect some vulnerabilities: unsafe approximation :-(
  - or we declare certain good programs as vulnerable: safe approximation :-)
  - or both :-((

- tradeoff between efficiency of analysis and precision of analysis.

Use of integer analysis

Most vulnerabilities are caused due to improper string manipulation.

Modify the program to include

- integer variables representing lengths of strings, overlaps between strings, etc.
- safety conditions before all string manipulation instructions.

Use well-known integer analysis algorithms to verify the safety conditions.

$\Longrightarrow$ we reduce string analysis problem to integer analysis problem :-)

Ideas: Dor, Rodeh and Sagiv

Original C code

```
char s[10];
s[15] = ’a’;
```

Instrumented C code

```
char s[10];  int sAlloc = 10;
             assert (15 < sAlloc);
s[15] = ’a’;
```

The integer variable sAlloc remembers the space allocated for string s.

The statement assert(15 < sAlloc); says that the program should abort here if $\text{sAlloc} \leq 15$.

We use an integer analysis algorithm to check that the assert conditions are satisfied.

Handling pointer arithmetic.

Original C code

```
char s [10];
char *p;
p = s + 7;
p[5] = ’a’;
```

Handling pointer arithmetic.

Original C code

```
char s [10];
char *p;
p = s + 7;
p [5]  = ’a’;
```

Instrumented C code

```
char s [10];  int sAlloc = 10;
char *p;      int pAlloc = 0;
              assert (7 <= sAlloc);
p = s + 7;    pAlloc = sAlloc - 7;
              assert (5 < pAlloc);
p [5]  = ’a’;
```

The second assert condition does not hold, as desired.

Complex control flow constructs are automatically handled.

```
char s [10];
int i;
for (i=0; i<=15; i++) {
  s[i] = ’a’;
}
```

```
char s [10];  int sAlloc = 10;
int i;
for (i=0; i <=15; i++) {
            assert (i < sAlloc);
  s[i] = ’a’;
}
```

The asserted condition will be violated at some point during the execution of the program, as desired.

String manipulation functions like strcpy, strlen, strcat should be treated directly, without analyzing their code.

```
char s[10];
char t[10];
strcpy (s,t);
```

This code is vulnerable.

Cannot be detected from information about sAlloc and tAlloc.

Need further variables:

sIsNull	s is a null terminated string (boolean)
sLen	length of s

Instrumented code

```
char s [10];     int sAlloc=10, sIsNull=false, sLen;
char t [10];     int tAlloc=10, tIsNull=false, tLen;
                 assert (tIsNull && tLen < sAlloc)
strcpy (s,t);
                 sIsNull=true; sLen=tLen;
```

The asserted condition is violated, as desired.

```
char *p;            int pAlloc=0, pIsNull=false, pLen;
char s[20];         int sAlloc=20, sIsNull=false, sLen;
p=”Hello World!”;   pAlloc=13; pIsNull=true; pLen=12;
                    assert(pIsNull && pLen < sAlloc)
strcpy(s,p);
                    sIsNull=true; sLen=pLen;
```

The asserted condition holds, as desired.

Dealing with string overlaps.

```
char *p, *q, s [20], t [20];      ... instrumentation code ...
p=”Hello World!”;                 ...
q=s+6;                            ...
                    /* here qIsNull == sIsNull == false */
strcpy(s,p);                      sIsNull=true; sLen=pLen;
                  /* here sIsNull == true, qIsNull == false */
                                  assert (qIsNull && qLen < tAlloc)
strcpy(t,q);                      ...
```

The asserted condition for second strcpy fails :-(

After the first strcpy, the variables qIsNull and qLen are not updated.

Dealing with string overlaps.

```
char *p, *q, s[20], t[20];        ... instrumentation code ...
p=”Hello World!”;                 ...
q=s+6;                            ...
                  /* here qIsNull == sIsNull == false */
strcpy(s,p);                      sIsNull=true; sLen=pLen;
                /* here sIsNull == true, qIsNull == false */
                                  assert (qIsNull && qLen < tAlloc)
strcpy(t,q);                      ...
```

The asserted condition for second strcpy fails :-(

After the first strcpy, the variables qIsNull and qLen are not updated.

⟹ need further variables for keeping track of overlaps between strings.

Putting together

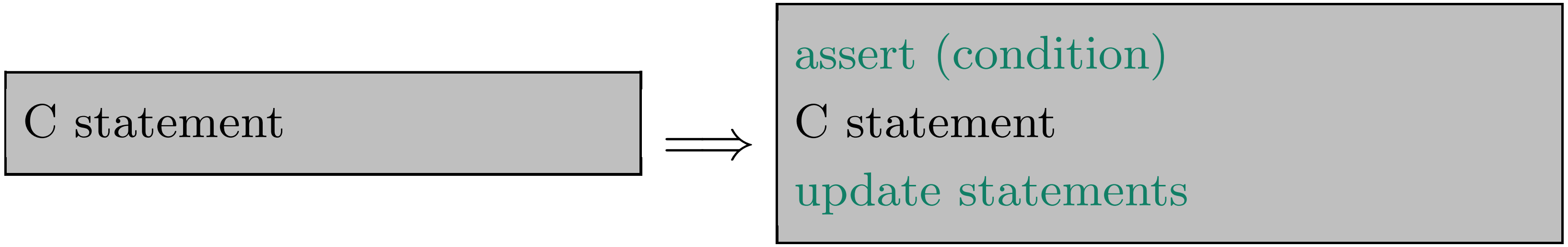
The required list of variables:

sAlloc	space allocated for string ccodes
sIsNull	whether string s is null terminated
sLen	length of string s
s_overlaps_t	whether strings s and t point inside the same allocated buffer
s_diff_t	amount of overlap between strings s and t

s_overlaps_t is same as t_overlaps_s.

s_diff_t = -t_diff_s.

Schema for instrumenting the C code.



C statement

$\Longrightarrow$

assert (condition)
C statement
update statements

Clean program: all the string operations have a well defined output (according to standard specifications.)

The instrumentation preserves the bahaviour of clean C programs.

In a program is unclean, the condition for the corresponding statement is violated at some time during execution.

Allocation

C statement	condition	update

char s [20];	true	sAlloc = 20;
		sIsNull = false;
		FOREACH a
		a_overlaps_s = false

No safety conditions required.

The string is not null-terminated and has no overlap with any other string.

Allocation

p = malloc(exp)	true

```
if (p)
   pAlloc = exp;
   else pAlloc = 0;
pIsNull = false;
FOREACH a
   a_overlaps_p = false;
```

If allocation fails then no space is allocated for the string.

Constant string assignment

s = ”some string”;

true

```
sAlloc = 12;
sIsNull = true;
sLen = 11;
FOREACH a
   s_overlaps_a = false;
```

No assertion conditions.

The string is null terminated and has no overlap with other strings.

Safe even with other pointers to the same string constant, as no updates are allowed in this region of the memory.

Pointer arithmetic For simplicity consider only exp ≥ 0

C statement	condition
`p = q + exp;`	`exp <= qAlloc`

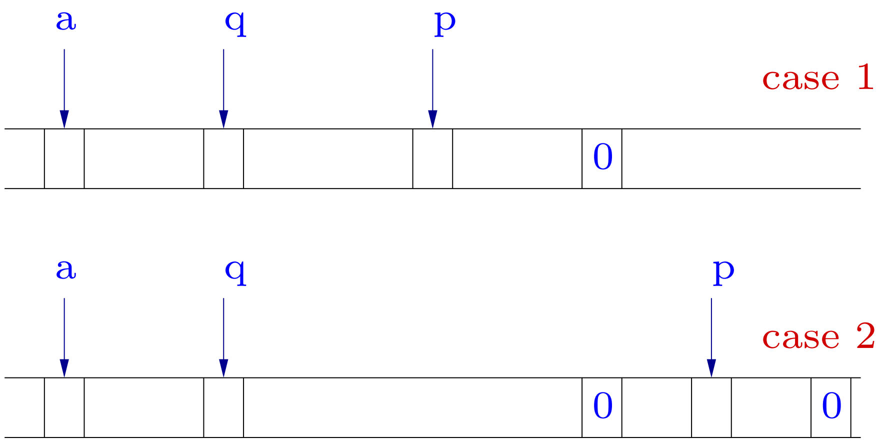
update

```
pAlloc = qAlloc - exp;

p_overlaps_q = true; p_diff_q = exp;

FOREACH a
   p_overlaps_a = q_overlaps_a;
   p_diff_a = q_diff_a + exp;
...
```

```
...
if (qIsNull && qLen >= exp) {
   pIsNull = true; pLen = qLen - exp;
} else RECOMPUTE (p);
```



```
#define RECOMPUTE (s)
   sLen = strlen(s);
   sIsNull = (sLen < sAlloc ? true : false)
/* however strlen cannot be analyzed precisely! */
```

String update We consider only $i \geq 0$

C statement

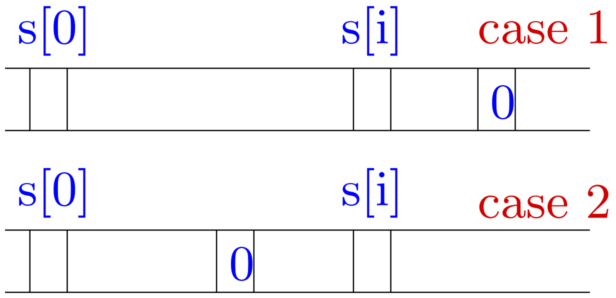
```
s[i] = exp;
```

condition

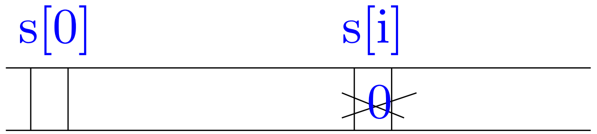
```
i < sAlloc
```

Update

```
if (exp == 0) {
   if (!sIsNull || sLen > i) {
      sIsNull = true;
      sLen = i;
   }
   FOREACH a
      DESTRUCTIVE_UPDATE (a,s)
}
```



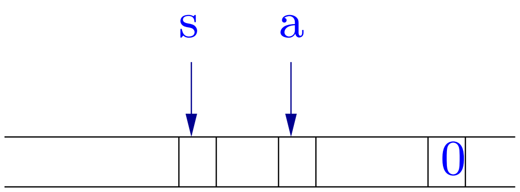
```
else {
    if (sIsNull && i == sLen)
        RECOMPUTE (s);
    FOREACH a
        DESTRUCTIVE_UPDATE (a,s);
}
```

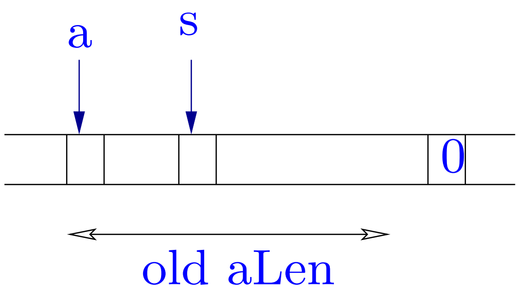


DESTRUCTIVE_UPDATE

The string `s` has been modified and variables `sIsNull` and `sLen` have been updated. The corresponding variables for overlapping strings need to be updated.

```
#define DESTRUCTIVE_UPDATE (a,s)
  if (a_overlaps_s)
    if (sIsNull && a_diff_s <= sLen &&
        (!aIsNull || a_diff_s >= −aLen)) {
      aIsNull = true;
      aLen = sLen − a_diff_s;
    } else RECOMPUTE (a);
```





Library functions: strcpy

C statement

```
strcpy (s,t);
```

condition

tIsNull & tLen < sAlloc

update

```
sIsNull = true;
sLen = tLen;
FOREACH a
  DESTRUCTIVE_UPDATE (a,s);
```

The copied string should be null terminated and the destination should have enough space.

Library functions: strcat

C statement

```
strcat (s,t);
```

condition

```
sIsNull && tIsNull
   && tLen + sLen < sAlloc
```

update

```
sLen = sLen + tLen;
FOREACH a
  DESTRUCTIVE_UPDATE (a,s);
```

Both the source and destination strings should be null terminated before concatenation.

Library functions: strcat

C statement

```
strcat (s,t);
```

condition

```
sIsNull && tIsNull
   && tLen + sLen < sAlloc
```

update

```
sLen = sLen + tLen;
FOREACH a
  DESTRUCTIVE_UPDATE (a,s);
```

Both the source and destination strings should be null terminated before concatenation.

Normal functions: to be discussed.

Given a C program, we have shown how to compute an instrumented C program which preserves the semantics.

If the original C program is clean then the instrumented C program has the same behaviour and all assertions always hold.

If the original C program has an unclean expression then the corresponding assertion will be false at some time.

Next, we use integer analysis algorithms to check whether any of the assertions are violated.

A program state at a certain point of time during the program execution tells us the value of each program variable at that time.

Execution of an instruction leads to a modification in the program state.

Each program point can be reached several times during execution (loops).

Hence several program states are possible at each program point.

Goal: for each program point, compute an upper approximation of the set of possible program states.

Upper approximation of the set of possible states is a safe approximation.

Scenario 1:

```
char s[20];
for (i=0; i<10; i++) {
  j = 2 * i;
  /* j is hopefully < 20 */
  s[j] = 'a';
}
```

The possible values of (i, j) before the string update operation are

$$(0,0),(1,2),(2,4)...(9,18)$$

Suppose our analysis tells us that at this program point:

$0 \leq i \leq 9 \wedge 0 \leq j \leq 18$ upper approximation

We conclude that the program is clean safe

Upper approximation of the set of possible states is a safe approximation.

Scenario 2:

```
char s [20];
for (i=0; i<10; i++) {
  j = 2 * i;
  /* j is hopefully < 20 */
  s[j] = 'a';
}
```

The possible values of (i, j) before the string update operation are

$$(0,0),(1,2),(2,4)...(9,18)$$

Suppose our analysis tells us that at this program point:

$0 \leq i < \infty \wedge 0 \leq j < \infty$ upper approximation

We conclude that the program is not clean safe

Upper approximation of the set of possible states is a safe approximation.

Scenario 3:

```
char s[20];
for (i=0; i<=10; i++) {
  j = 2 * i;
  /* j is hopefully < 20 */
  s[j] = 'a';
}
```

The possible values of (i, j) before the string update operation are

$$(0,0),(1,2),(2,4)...(10,20)$$

We compute upper approximation of the set of possible states.

Hence our analysis should always tell us that j can become 20.

We conclude that the program is not clean safe

We transform the instrumented program to a program with only integer variables $\Longrightarrow$ further safe approximation.

e1 is non-integer variable:

$$\text{e1} = \text{e2}; \Longrightarrow \ ;$$

e contains non-integer variables and constants:

$$\text{x} = \text{e}; \Longrightarrow \text{x} = ?;$$
$$\text{if (e) s1 else s2} \Longrightarrow \text{if (?) s1 else s2}$$

The expression ? can take all possible values non-deterministically. (In practice, use a special uninitialized variable in its place.)

Safe approximation: all executions of the original program are still allowed after approximation.

Instrumented program

```
char s [20];     int sAlloc=20, sIsNull=false, sLen;
for (i=0; i<=10; i++) {
  j = 2 * i;     assert (sAlloc > j)
  s[j] = ’a’;    if (97 == 0)  ...
}
```

Instrumented program

```
char s [20];     int sAlloc=20, sIsNull=false, sLen;
for (i=0; i<=10; i++) {
  j = 2 * i;     assert (sAlloc > j)
  s[j] = ’a’;    if (97 == 0)  ...
}
```

Corresponding integer program

```
               int sAlloc=20, sIsNull=false, sLen;
for (i=0; i<=10; i++) {
  j = 2 * i;   assert (sAlloc > j)
               if (97 == 0)  ...
}
```

This may involve some safe approximation

Instrumeted program:

```
char s [10], *t; ...
t = ”Hello!”;     tAlloc = 7; tIsNull = 0; tLen=6; ...
strcpy (s,t);     ...sLen=tLen
 if (s[0]==72) i = 5; else i = 6;
s[i] = 0;         if (0==0) if (!sIsNull || sLen > i) {
                        sIsNull=true; sLen=i;}
```

This may involve some safe approximation

Instrumeted program:

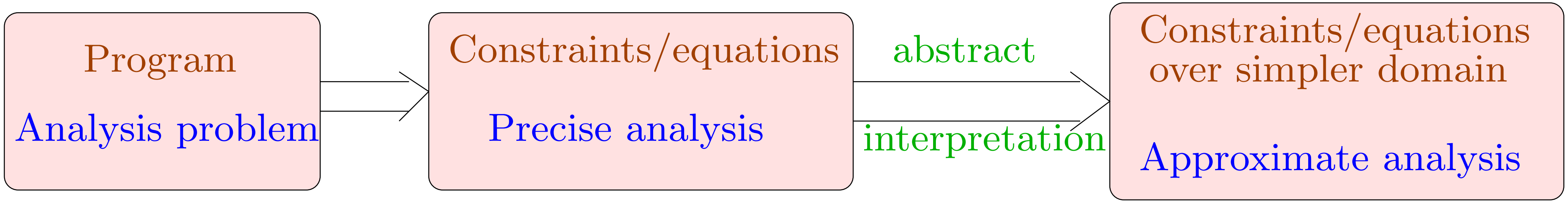
```
char s[10], *t;  ...
t = "Hello!";    tAlloc = 7; tIsNull = 0; tLen=6; ...
strcpy(s,t);     ...sLen=tLen
 if (s[0]==72) i = 5; else i = 6;
s[i] = 0;        if (0==0) if (!sIsNull || sLen > i) {
                     sIsNull=true; sLen=i;}
```

Integer program:

```
                 ... int any;
                 tAlloc = 7; tIsNull = 0; tLen=6; ...
                 ...sLen=tLen
 if (any      ) i = 5; else i = 6;
s[i] = 0;        if (0==0) if (!sIsNull || sLen > i) {
                     sIsNull=true; sLen=i;}
```

Program analysis for integers relations

Our methodology:



Precise analysis:	what values are taken by variable x at a certain program point?	infinite domain: $\mathbb{Z}$
Approximate analysis:	does variable x ever take a negative value at a certain program point?	finite domain: $\{+,-,0\}$

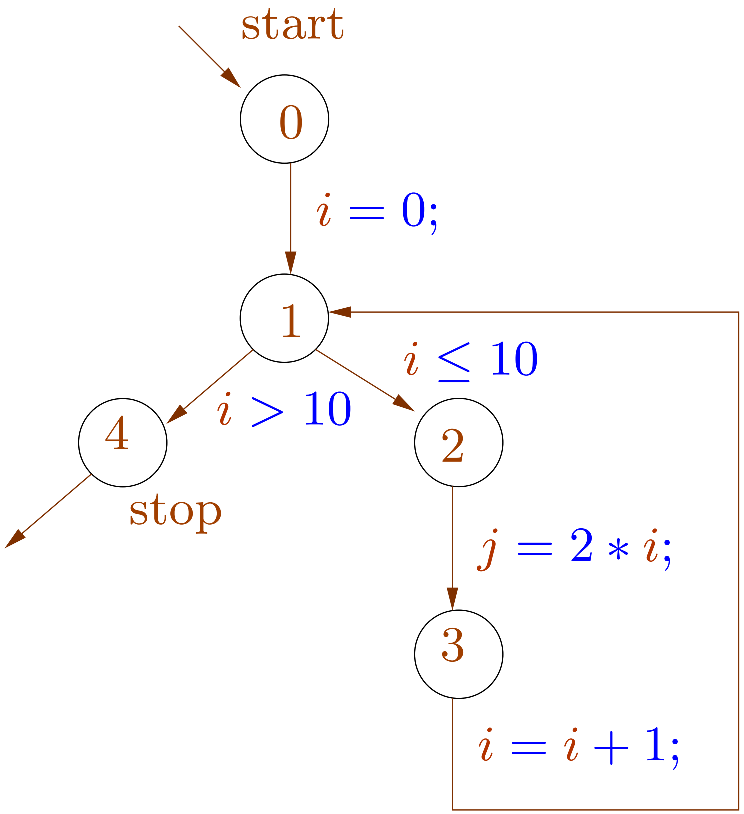
We consider a set Vars of variables ranging over integers.

Program consists of statements of the form

NOP	;
Assignments	x = e;
Conditions	if (e) s1 else s2
Jumps	goto L

While and for loops: translated using conditions and goto statements.

We represent programs using control flow graphs (CFGs).



Distinguished *start* and *stop* nodes.

Edges k are of the form (u, l, v) where u and v are nodes and label l is an assignment or a condition.

The set of possible states state of the program is

$$\mathcal{S} = \mathsf{Vars} \to \mathbb{Z}$$

The evaluation of an arithmetic expression e under state $\rho \in \mathcal{S}$ is denoted

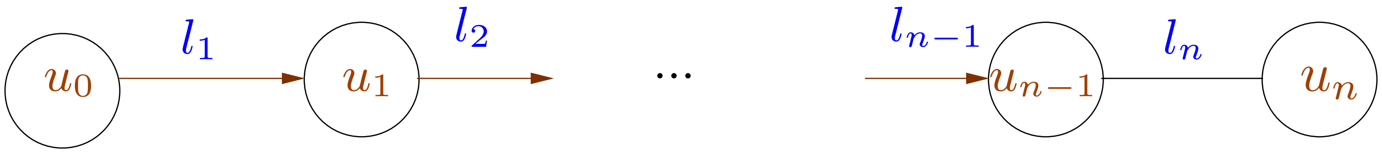
$$[\![e]\!]\ \rho : \mathbb{Z}$$

An edge $k = (u, l, v)$ induces a partial transformation on program states. The transformation depends only on the label l.

$$[\![k]\!]\ \rho = [\![l]\!]\ \rho$$
$$\text{where } [\![l]\!]\ : \mathcal{S} \to \mathcal{S}$$

$$\begin{aligned}
[\![;]\!]\ \rho &= \rho; \\
[\![x = e;]\!]\ \rho &= \rho \oplus \{x \mapsto [\![e]\!]\ \rho\} \\
[\![e_1 \geq e_2]\!]\ \rho &= \rho \qquad\qquad \text{if } [\![e_1]\!]\ \rho \geq [\![e_2]\!]\ \rho
\end{aligned}$$

A path π is a sequence of consequetive edges in the CFG.



$$\pi = k_1, \ldots, k_n \text{ where each } k_i \text{ is of the form } (u_{i-1}, l_i, u_i).$$

$$\text{We write } \pi : u_0 \rightarrow^* u_n$$

The transformation induced by a path is the composition of the transformations induced by the edges.

$$[\![\pi]\!] \; = [\![k_n]\!] \circ \ldots \circ [\![k_1]\!]$$

Each node can be reached through possibly infinitely many paths, leading to infinitely many different states at each program point.

We are interested in the set of all such states at each program point.

Suppose we know that a set V of states is possible at a node u.

By following an edge $k = (u, , v)$, a new set of states becomes possible at node v. This set is denoted $[\![k]\!]^\sharp\ V = [\![l]\!]^\sharp\ V : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$.

We define abstract transformation

$$[\![l]\!]^\sharp\ V = \{[\![l]\!]\ \rho \mid \rho \in V \text{ and } [\![l]\!]\ \text{ is defined for } \rho\}.$$

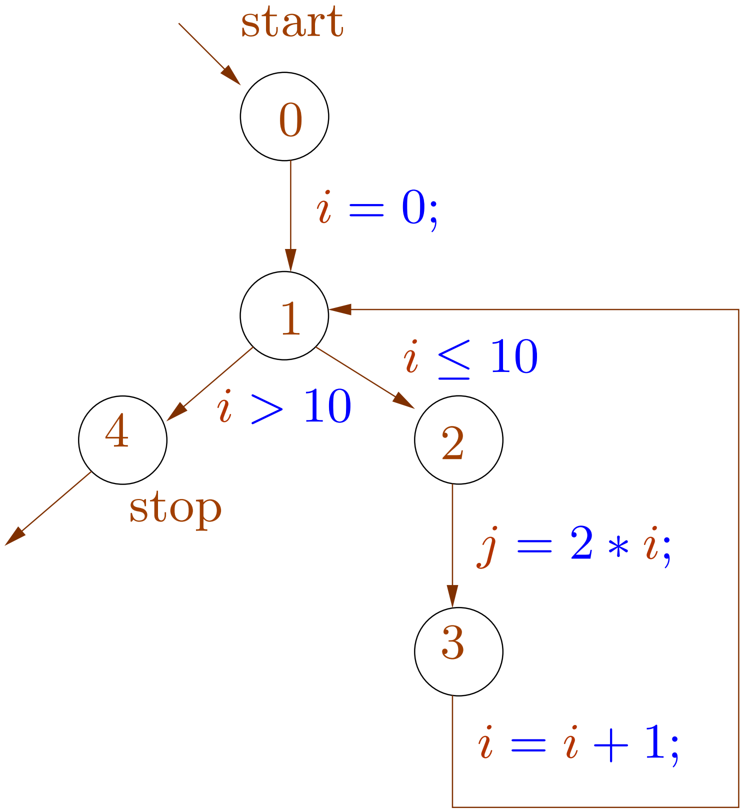
As before, $[\![k_1, \ldots, k_n]\!]^\sharp\ V = ([\![k_n]\!]^\sharp\ \circ \ldots \circ [\![k_1]\!]^\sharp\ )V$.

At the *start* node, all states are possible.

For each node v we want to compute the set

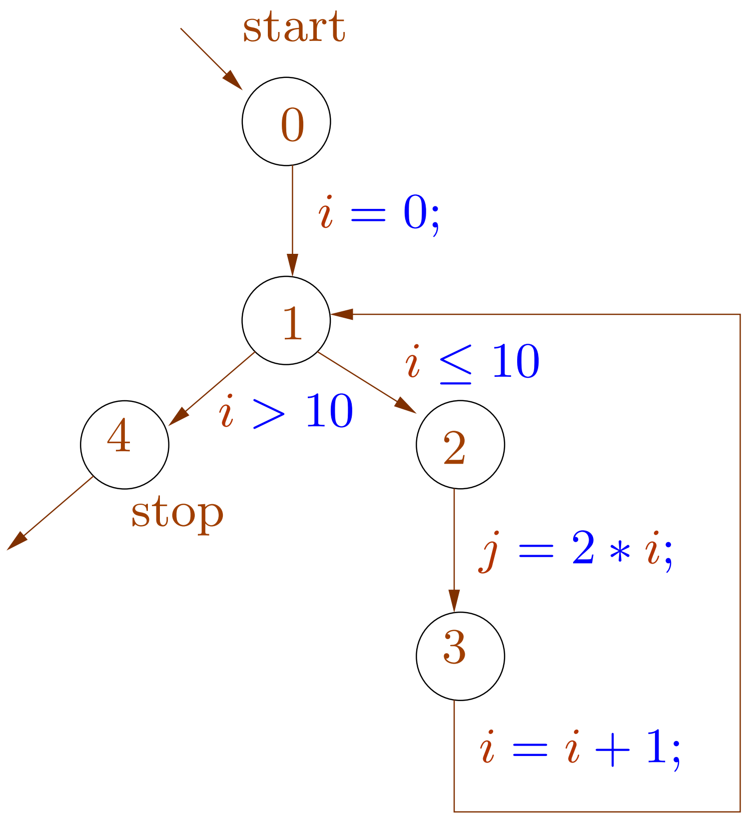
$$\mathcal{V}^*[v] = \bigcup\{[\![\pi]\!]^\sharp\ \mathcal{S} \mid \pi : \mathit{start} \to^* v\}$$

Example



u	$\mathcal{V}^*[u]$
0	$-\infty < i, j < \infty$
1	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 11 \wedge j = 2i - 2$
2	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 10 \wedge j = 2i - 2$
3	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 10 \wedge j = 2i$
4	$i = 11 \wedge j = 20$

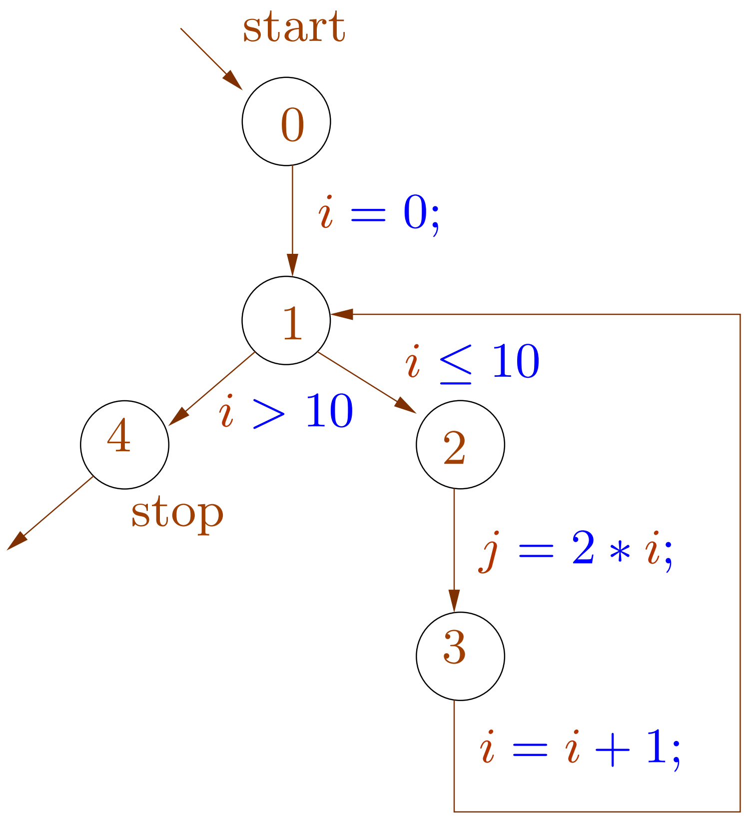
Example



u	$\mathcal{V}^*[u]$
0	$-\infty < i, j < \infty$
1	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \le i \le 11 \wedge j = 2i{-}2$
2	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \le i \le 10 \wedge j = 2i{-}2$
3	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \le i \le 10 \wedge j = 2i$
4	$i = 11 \wedge j = 20$

How to compute the sets $\mathcal{V}^*[v]$ in general?

Example

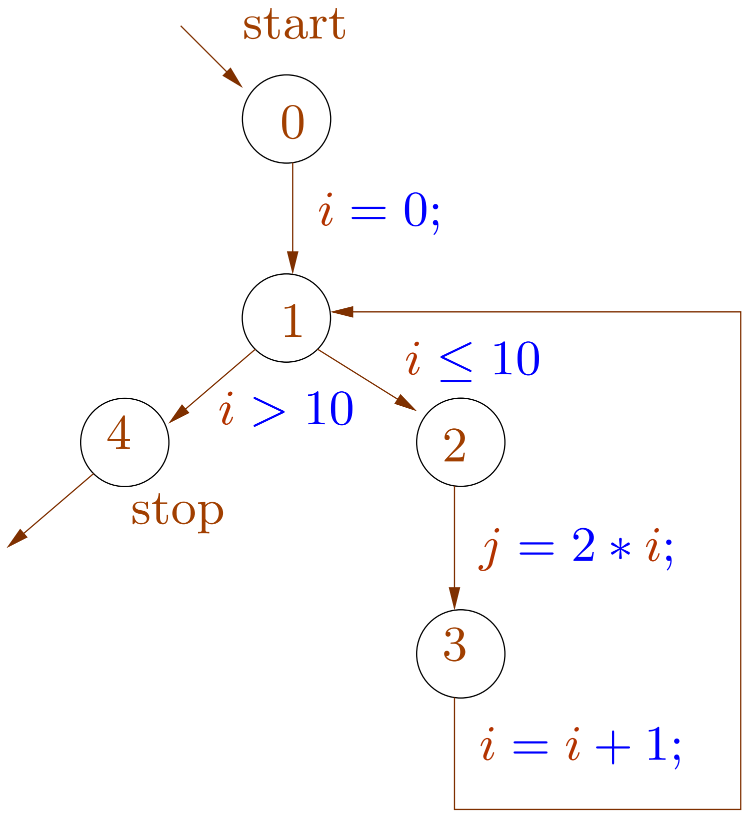


u	$\mathcal{V}^*[u]$
0	$-\infty < i, j < \infty$
1	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 11 \wedge j = 2i - 2$
2	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 10 \wedge j = 2i - 2$
3	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 10 \wedge j = 2i$
4	$i = 11 \wedge j = 20$

How to compute the sets $\mathcal{V}^*[v]$ in general?

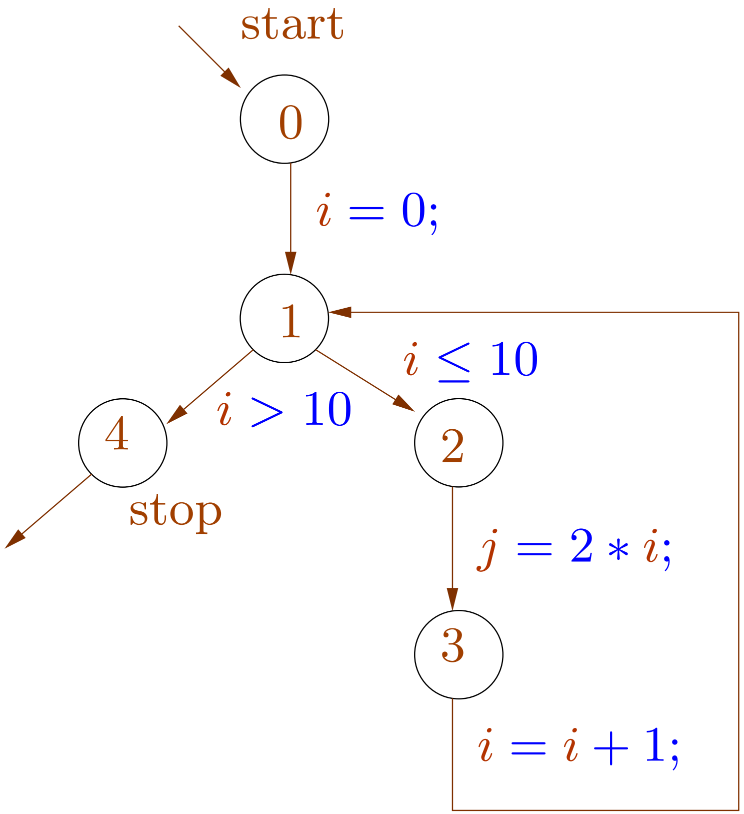
In general they are not computable!

We set up a constraint system.



$$
\begin{aligned}
\mathcal{V}[0] &\supseteq \quad \mathcal{S} \\
\mathcal{V}[1] &\supseteq \quad [\![i = 0;]\!]\ \mathcal{V}[0] \\
\mathcal{V}[1] &\supseteq \quad [\![i = i{+}1;]\!]\ \mathcal{V}[0] \\
\mathcal{V}[2] &\supseteq \quad [\![i \leq 10]\!]\ \mathcal{V}[1] \\
\mathcal{V}[3] &\supseteq \quad [\![j = 2{*}i;]\!]\ \mathcal{V}[0] \\
\mathcal{V}[4] &\supseteq \quad [\![i > 10]\!]\ \mathcal{V}[1]
\end{aligned}
$$

We set up a constraint system.



$$
\begin{aligned}
\mathcal{V}[0] &\supseteq \quad \mathcal{S} \\
\mathcal{V}[1] &\supseteq \quad [\![i = 0;]\!] \; \mathcal{V}[0] \\
\mathcal{V}[1] &\supseteq \quad [\![i = i{+}1;]\!] \; \mathcal{V}[0] \\
\mathcal{V}[2] &\supseteq \quad [\![i \leq 10]\!] \; \mathcal{V}[1] \\
\mathcal{V}[3] &\supseteq \quad [\![j = 2{*}i;]\!] \; \mathcal{V}[0] \\
\mathcal{V}[4] &\supseteq \quad [\![i > 10]\!] \; \mathcal{V}[1]
\end{aligned}
$$

The least solution (wrt $\subseteq$) of the constraints is exactly $\mathcal{V}^*$.

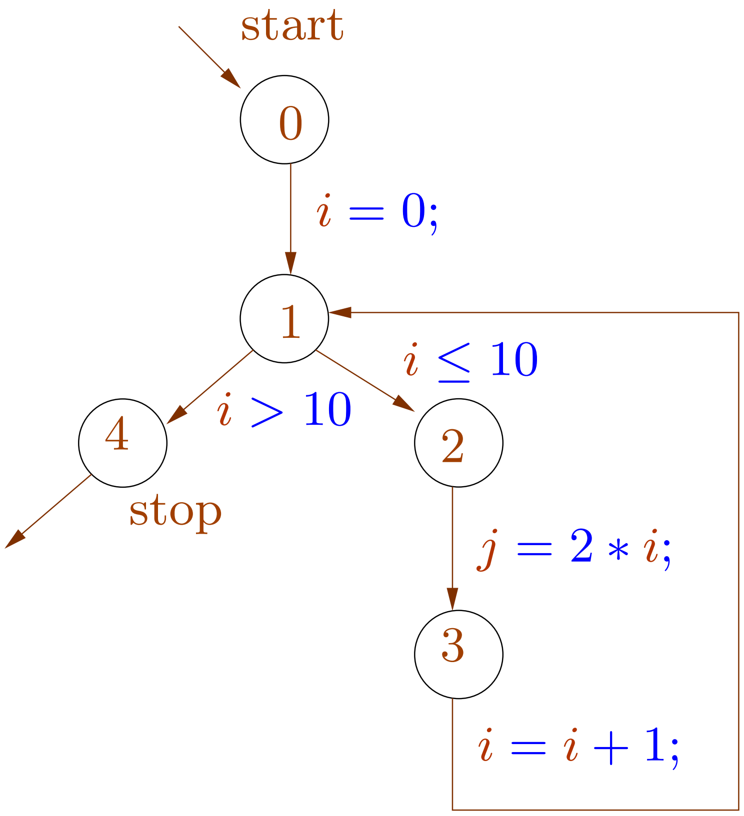
The least solution (wrt $\subseteq$) of the constraints is exactly $\mathcal{V}^*$.

Is this always true?

Does such a constraint system always have a least solution?

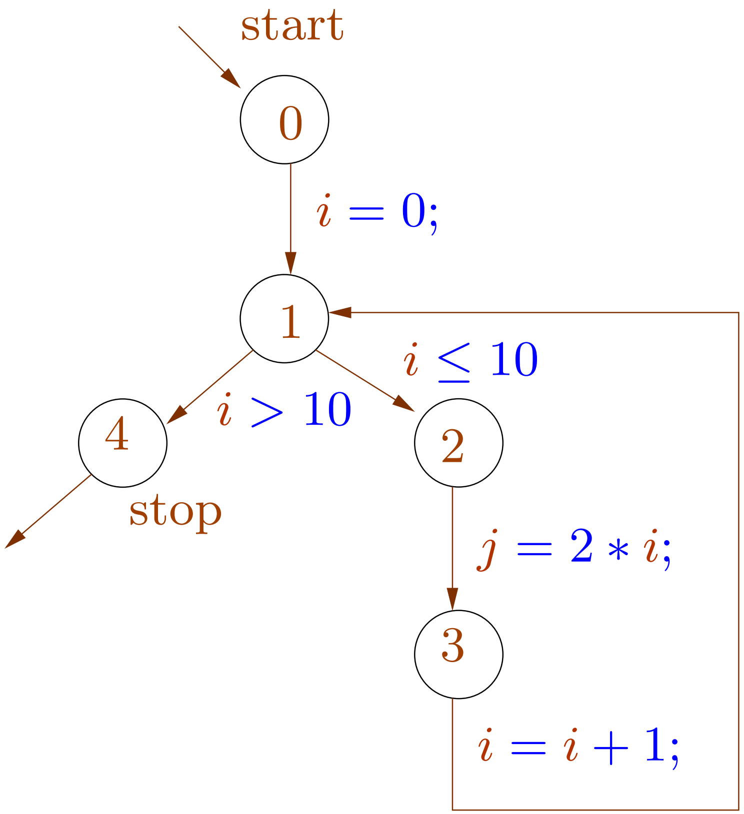
Is it computable? Efficiently?

An idea: do iterative computation of reachable states.



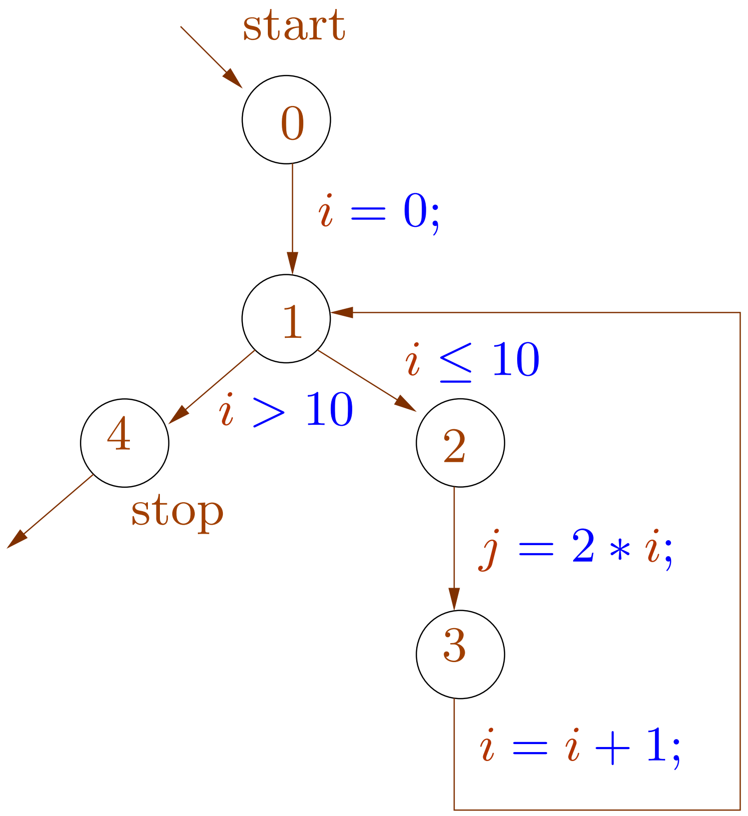
$\mathcal{V}[0]$	$\emptyset$
$\mathcal{V}[1]$	$\emptyset$
$\mathcal{V}[2]$	$\emptyset$
$\mathcal{V}[3]$	$\emptyset$
$\mathcal{V}[4]$	$\emptyset$

An idea: do iterative computation of reachable states.



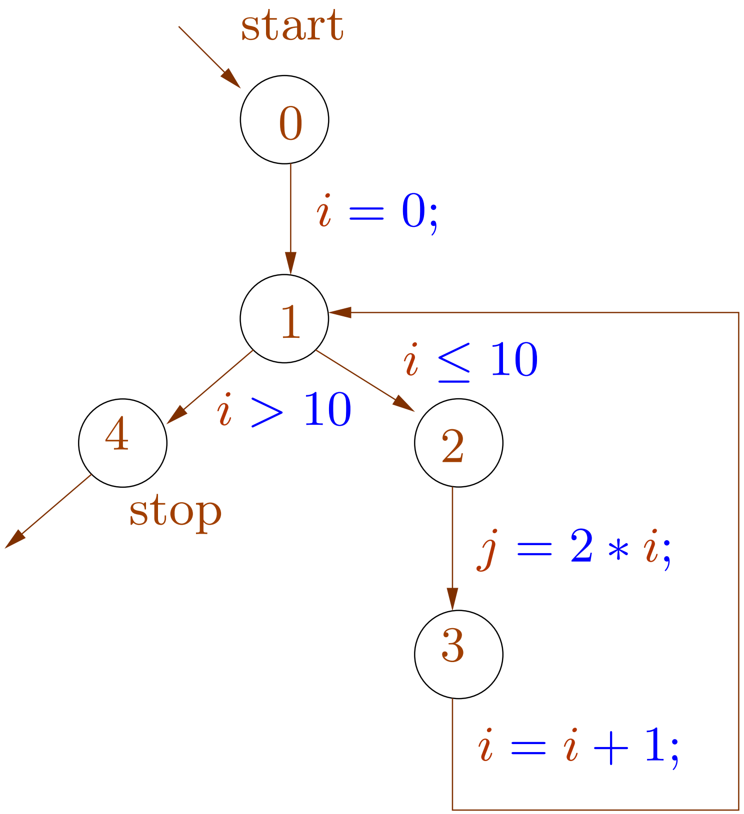
$\mathcal{V}[0]$	$\emptyset$	$\mathbb{Z} \times \mathbb{Z}$
$\mathcal{V}[1]$	$\emptyset$	
$\mathcal{V}[2]$	$\emptyset$	
$\mathcal{V}[3]$	$\emptyset$	
$\mathcal{V}[4]$	$\emptyset$	

An idea: do iterative computation of reachable states.



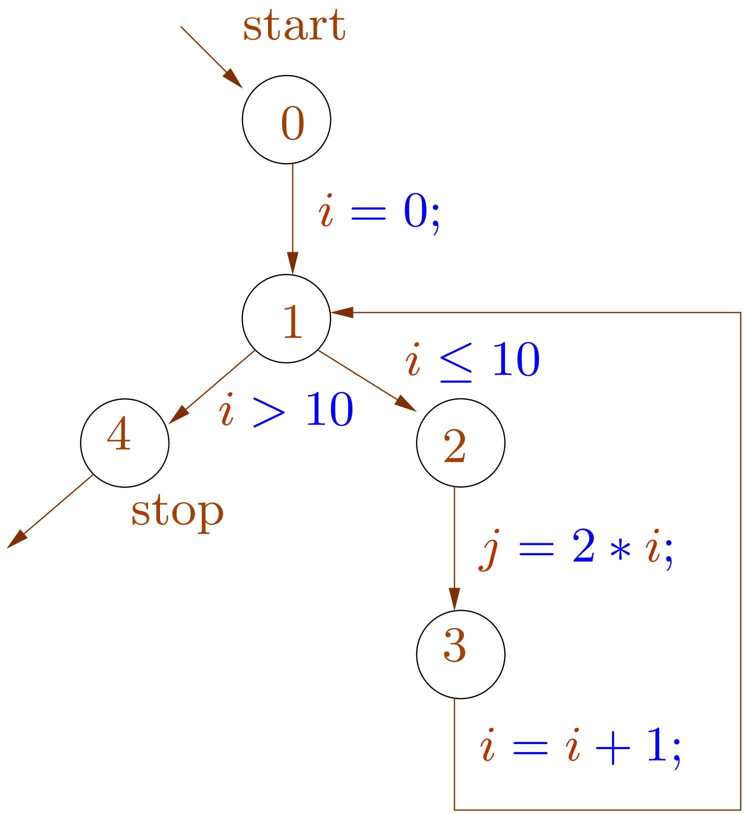
$\mathcal{V}[0]$	$\emptyset$	$\mathbb{Z} \times \mathbb{Z}$
$\mathcal{V}[1]$	$\emptyset$	$\{0\} \times \mathbb{Z}$
$\mathcal{V}[2]$	$\emptyset$	
$\mathcal{V}[3]$	$\emptyset$	
$\mathcal{V}[4]$	$\emptyset$	

An idea: do iterative computation of reachable states.



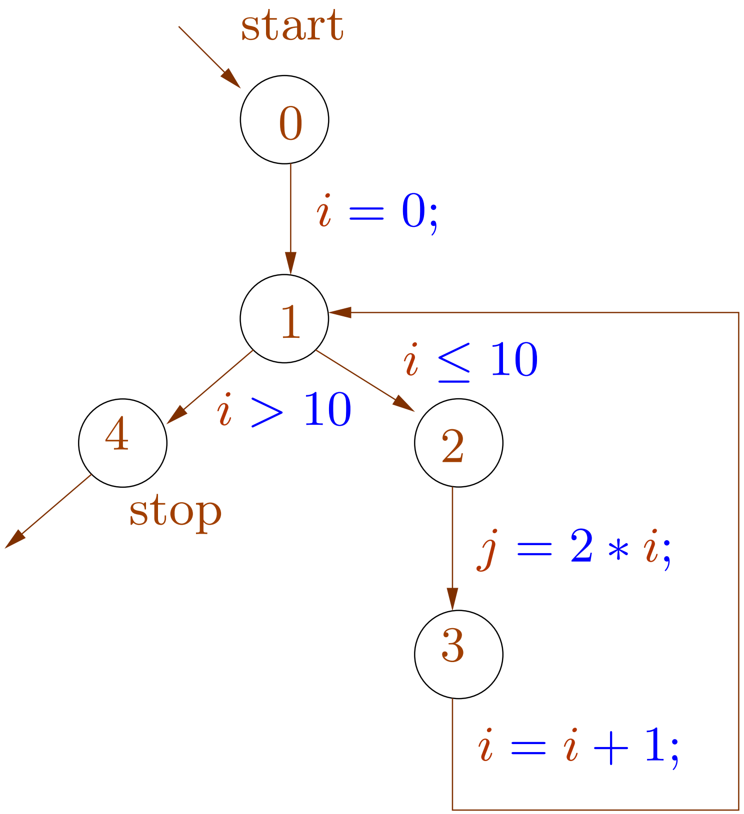
$\mathcal{V}[0]$	$\emptyset$	$\mathbb{Z} \times \mathbb{Z}$
$\mathcal{V}[1]$	$\emptyset$	$\{0\} \times \mathbb{Z}$
$\mathcal{V}[2]$	$\emptyset$	$\{0\} \times \mathbb{Z}$
$\mathcal{V}[3]$	$\emptyset$	
$\mathcal{V}[4]$	$\emptyset$	

An idea: do iterative computation of reachable states.



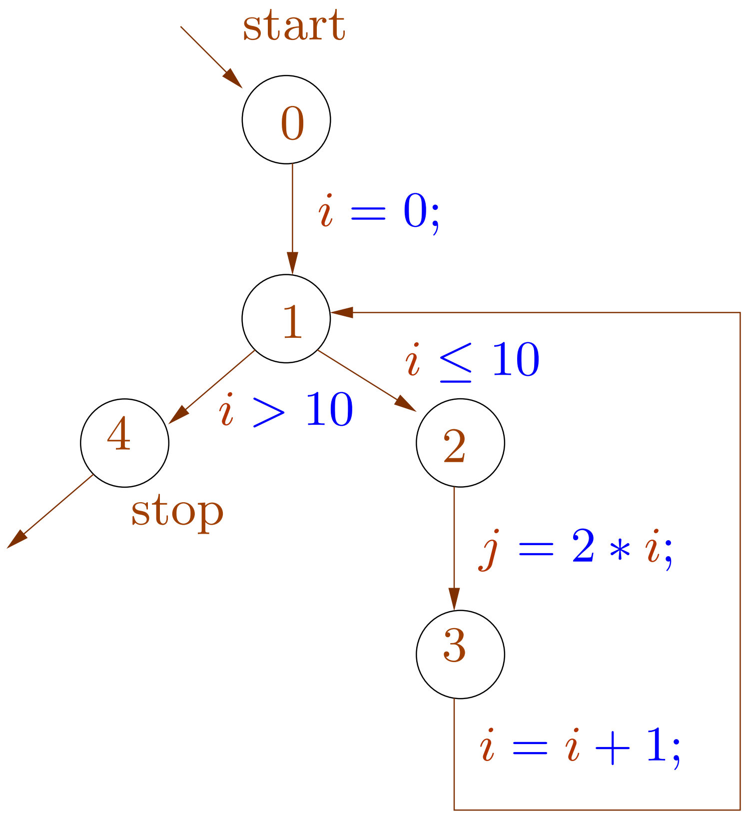
$\mathcal{V}[0]$	$\emptyset$	$\mathbb{Z} \times \mathbb{Z}$
$\mathcal{V}[1]$	$\emptyset$	$\{0\} \times \mathbb{Z}$
$\mathcal{V}[2]$	$\emptyset$	$\{0\} \times \mathbb{Z}$
$\mathcal{V}[3]$	$\emptyset$	$\{(0, 0)\}$
$\mathcal{V}[4]$	$\emptyset$	

An idea: do iterative computation of reachable states.



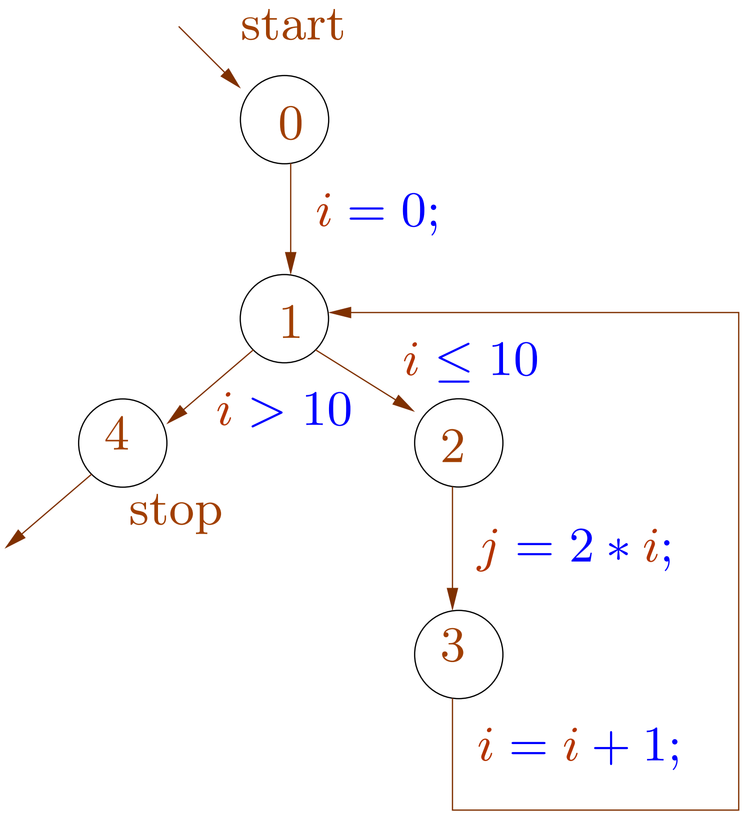
$\mathcal{V}[0]$	$\emptyset$	$\mathbb{Z} \times \mathbb{Z}$	
$\mathcal{V}[1]$	$\emptyset$	$\{0\} \times \mathbb{Z}$	$\{0, 1\} \times \mathbb{Z}$
$\mathcal{V}[2]$	$\emptyset$	$\{0\} \times \mathbb{Z}$	
$\mathcal{V}[3]$	$\emptyset$	$\{(0, 0)\}$	
$\mathcal{V}[4]$	$\emptyset$		

An idea: do iterative computation of reachable states.



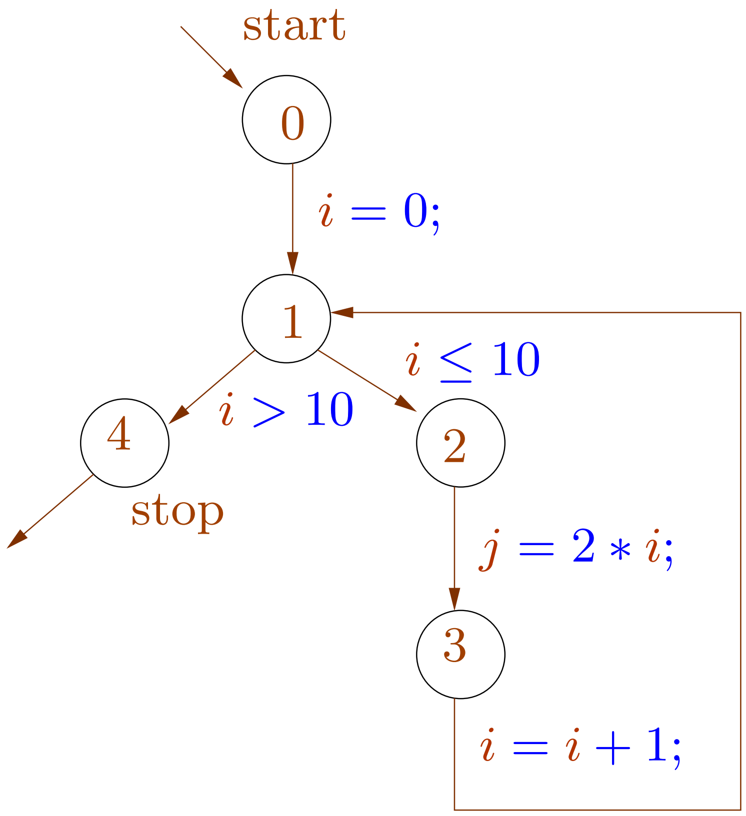
$\mathcal{V}[0]$	$\emptyset$	$\mathbb{Z} \times \mathbb{Z}$	
$\mathcal{V}[1]$	$\emptyset$	$\{0\} \times \mathbb{Z}$	$\{0, 1\} \times \mathbb{Z}$
$\mathcal{V}[2]$	$\emptyset$	$\{0\} \times \mathbb{Z}$	$\{0, 1\} \times \mathbb{Z}$
$\mathcal{V}[3]$	$\emptyset$	$\{(0, 0)\}$	
$\mathcal{V}[4]$	$\emptyset$		

An idea: do iterative computation of reachable states.



$\mathcal{V}[0]$	$\emptyset$	$\mathbb{Z} \times \mathbb{Z}$	
$\mathcal{V}[1]$	$\emptyset$	$\{0\} \times \mathbb{Z}$	$\{0, 1\} \times \mathbb{Z}$
$\mathcal{V}[2]$	$\emptyset$	$\{0\} \times \mathbb{Z}$	$\{0, 1\} \times \mathbb{Z}$
$\mathcal{V}[3]$	$\emptyset$	$\{(0, 0)\}$	$\{(0, 0), (1, 2)\}$
$\mathcal{V}[4]$	$\emptyset$		

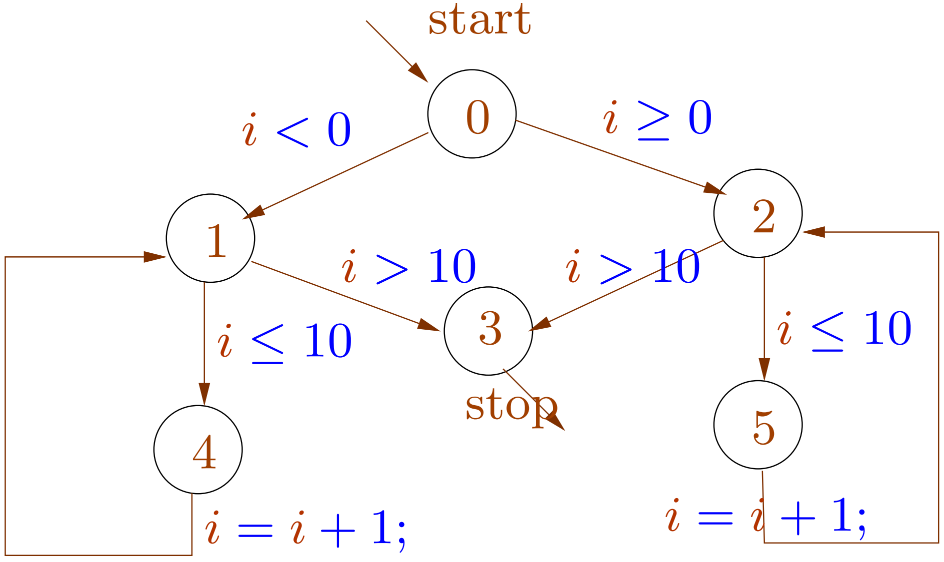
An idea: do iterative computation of reachable states.



$\mathcal{V}[0]$	$\emptyset$	$\mathbb{Z} \times \mathbb{Z}$		
$\mathcal{V}[1]$	$\emptyset$	$\{0\} \times \mathbb{Z}$	$\{0, 1\} \times \mathbb{Z}$	
$\mathcal{V}[2]$	$\emptyset$	$\{0\} \times \mathbb{Z}$	$\{0, 1\} \times \mathbb{Z}$	$\ldots$
$\mathcal{V}[3]$	$\emptyset$	$\{(0, 0)\}$	$\{(0, 0), (1, 2)\}$	
$\mathcal{V}[4]$	$\emptyset$			

Problem: too many iterations, infinite loops.

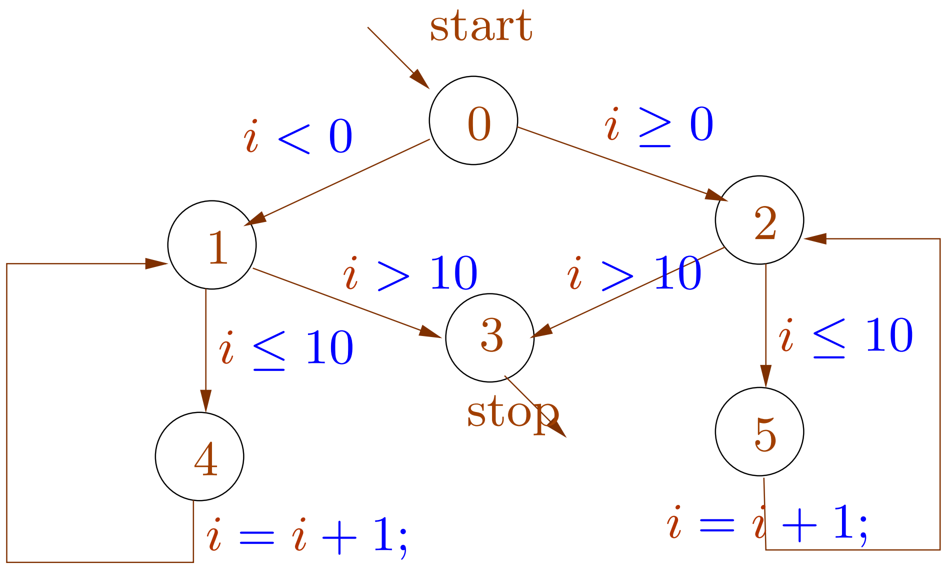
Solution: approximate computation of possible states.



0	$\emptyset$	$\mathbb{Z}$	$\mathbb{Z}$
1	$\emptyset$	$\mathbb{Z}^-$	$\mathbb{Z}$
2	$\emptyset$	$\mathbb{Z}^+$	$\mathbb{Z}^+$
3	$\emptyset$	$\mathbb{Z}^+$	$\mathbb{Z}^+$
4	$\emptyset$	$\mathbb{Z}^-$	$\mathbb{Z}$
5	$\emptyset$	$\mathbb{Z}^+$	$\mathbb{Z}^+$

Problem: too many iterations, infinite loops.

Solution: approximate computation of possible states.



0	$\emptyset$	$\mathbb{Z}$	$\mathbb{Z}$
1	$\emptyset$	$\mathbb{Z}^-$	$\mathbb{Z}$
2	$\emptyset$	$\mathbb{Z}^+$	$\mathbb{Z}^+$
3	$\emptyset$	$\mathbb{Z}^+$	$\mathbb{Z}^+$
4	$\emptyset$	$\mathbb{Z}^-$	$\mathbb{Z}$
5	$\emptyset$	$\mathbb{Z}^+$	$\mathbb{Z}^+$

Interpretation of our result:

the values of i at node 1 is included in $\mathbb{Z}$

the values of i at node 2 is included in $\mathbb{Z}^+$

This information we obtain is accurate.

In general we have some domain $\mathbb{D}$.

Examples: $2^{\mathcal{S}}$, $2^{\mathbb{Z}}$, $\{\emptyset, \mathbb{Z}^-, \mathbb{Z}^+, \mathbb{Z}\}$, the set of intervals over $\mathbb{Z}$.

In general we have some domain $\mathbb{D}$.

Examples: $2^{\mathcal{S}}$, $2^{\mathbb{Z}}$, $\{\emptyset, \mathbb{Z}^-, \mathbb{Z}^+, \mathbb{Z}\}$, the set of intervals over $\mathbb{Z}$.

We require an ordering $\sqsubseteq$ on the elements of this domain.

$\emptyset \sqsubseteq \mathbb{Z}^-$ $\quad$ $\emptyset \sqsubseteq \mathbb{Z}^+$ $\quad$ $\mathbb{Z}^- \sqsubseteq \mathbb{Z}$ $\quad$ $\mathbb{Z}^+ \sqsubseteq \mathbb{Z}$

Read $x \sqsubseteq y$ as "y is imprecise information compared to x".

In general we have some domain $\mathbb{D}$.

Examples: $2^{\mathcal{S}}$, $2^{\mathbb{Z}}$, $\{\emptyset, \mathbb{Z}^-, \mathbb{Z}^+, \mathbb{Z}\}$, the set of intervals over $\mathbb{Z}$.

We require an ordering $\sqsubseteq$ on the elements of this domain.

$\emptyset \sqsubseteq \mathbb{Z}^- \qquad \emptyset \sqsubseteq \mathbb{Z}^+ \qquad \mathbb{Z}^- \sqsubseteq \mathbb{Z} \qquad \mathbb{Z}^+ \sqsubseteq \mathbb{Z}$

Read $x \sqsubseteq y$ as "y is imprecise information compared to x".

We further require operations like least upper bounds.

$$\mathbb{Z}^- \sqcup \mathbb{Z}^+ = \mathbb{Z}$$

Recall: a set $\mathbb{D}$ with relation $\sqsubseteq$ is a partial order if the following conditions hold for all $x, y, z \in \mathbb{D}$.

- Reflexivity: $x \sqsubseteq x$.
- Antisymmetry: $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$.
- Transitivity: if $x \sqsubseteq y$ and $y \sqsubseteq z$ then $x \sqsubseteq z$.

An element $d \in \mathbb{D}$ is called an upper bound of a set $X \subseteq D$ if $x \sqsubseteq d$ for all $x \in X$.

$d \in \mathbb{D}$ is called least upper bound of $X \subseteq D$ if

- d is an upper bound of X
- $d \sqsubseteq d'$ for every upper bound d' of X

An element $d \in \mathbb{D}$ is called an upper bound of a set $X \subseteq D$ if $x \sqsubseteq d$ for all $x \in X$.

$d \in \mathbb{D}$ is called least upper bound of $X \subseteq D$ if

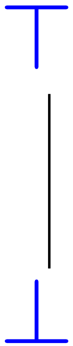
- d is an upper bound of X
- $d \sqsubseteq d'$ for every upper bound d' of X

A partial order $(\mathbb{D}, \sqsubseteq)$ is called a complete lattice if every $X \subseteq D$ has a least upper bound $\bigsqcup X$.

We write $x \sqcup y$ for $\bigsqcup\{x, y\}$.

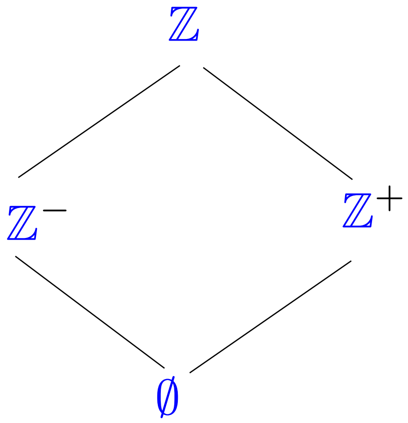
For $(2^{\mathcal{S}}, \subseteq)$ we have $\bigsqcup X = \bigcup X$.

Some complete lattices.



$\top$

$\bot$



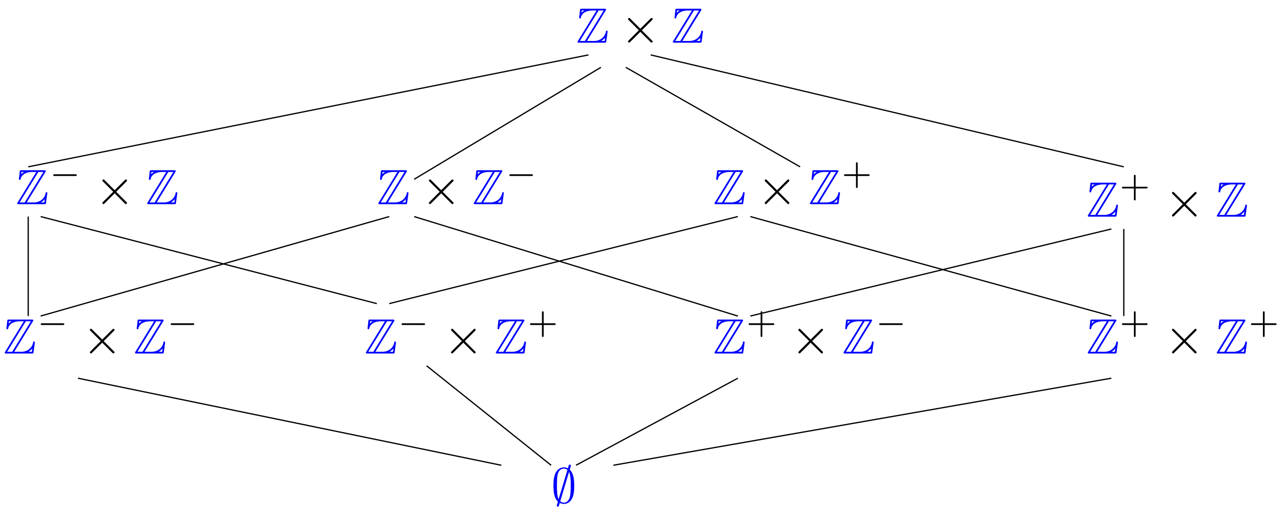
$\mathbb{Z}$

$\mathbb{Z}^-$ $\mathbb{Z}^+$

$\emptyset$

$$\mathbb{Z}^- = \{x \in \mathbb{Z} \mid x < 0\}$$

$$\mathbb{Z}^+ = \{x \in \mathbb{Z} \mid x \geq 0\}$$



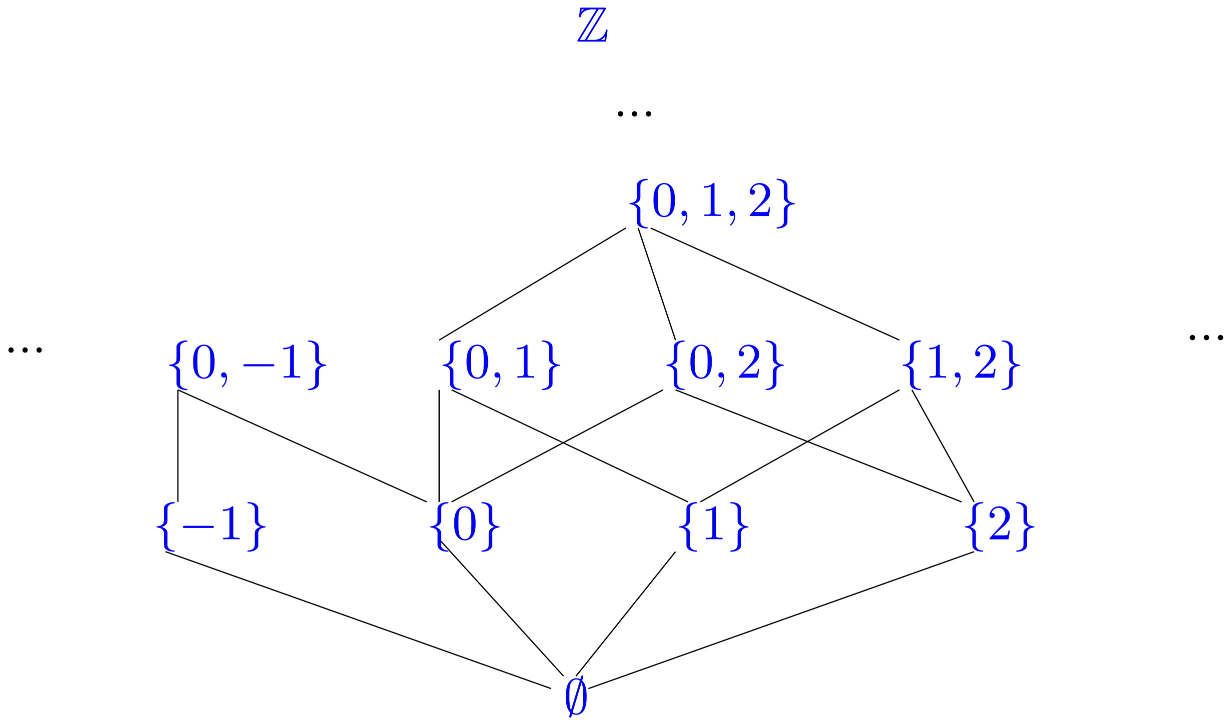
$\mathbb{Z} \times \mathbb{Z}$

$\mathbb{Z}^- \times \mathbb{Z}$ $\mathbb{Z} \times \mathbb{Z}^-$ $\mathbb{Z} \times \mathbb{Z}^+$ $\mathbb{Z}^+ \times \mathbb{Z}$

$\mathbb{Z}^- \times \mathbb{Z}^-$ $\mathbb{Z}^- \times \mathbb{Z}^+$ $\mathbb{Z}^+ \times \mathbb{Z}^-$ $\mathbb{Z}^+ \times \mathbb{Z}^+$

$\emptyset$

An infinite complete lattice : $(2^{\mathbb{Z}}, \subseteq)$.



Every complete lattice has

- a top element: $\top = \bigsqcup \mathbb{D}$
- a bottom element: $\bot = \bigsqcup \emptyset$

Further every $X \subseteq \mathbb{D}$ has a greatest lower bound $\bigsqcap X$.

For $(2^{\mathcal{S}}, \subseteq)$ we have $\bigsqcap X = \bigcap X$.

Consider the set of lower bounds of X:

$$L = \{l \in \mathbb{D} \mid \forall x \in X, l \leq x\}$$

and define

$$g = \bigsqcup L$$

Claim: g is the greatest lower bound of X.

(1)

g is a lower bound of X:

Consider any $x \in X$.

$l \leq x$ for all $l \in L$, i.e. x is an upper bound of L.

Hence $g = \bigsqcup L \sqsubseteq x$.

(2)

g is the greatest lower bound of X:

Let l be any other lower bound of X.

Then $l \in L$.

Hence $l \sqsubseteq \bigsqcup X = g$.

A function $f : \mathbb{D}_1 \longrightarrow \mathbb{D}_2$ is called monotone if:

$$f(x) \sqsubseteq f(y) \text{ whenever } x \sqsubseteq y$$

A function $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called monotone if:

$$f(x) \sqsubseteq f(y) \text{ whenever } x \sqsubseteq y$$

The function $f : \mathbb{Z} \to \mathbb{Z}$ defined as $f(x) = x + 1$ is monotone.

Note: $(\mathbb{Z}, \leq)$ is not a complete lattice.

A function $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called monotone if:

$$f(x) \sqsubseteq f(y) \text{ whenever } x \sqsubseteq y$$

The function $f : \mathbb{Z} \to \mathbb{Z}$ defined as $f(x) = x + 1$ is monotone.

Note: $(\mathbb{Z}, \leq)$ is not a complete lattice.

The transformations induced by the program edges are monotone:

$$\text{Recall: } [\![l]\!]^\sharp \; : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$$

$$[\![l]\!]^\sharp \; V = \{[\![l]\!] \; \rho \mid \rho \in V \text{ and } [\![l]\!] \; \text{ is defined for } \rho\}.$$

$$\text{Hence if } V_1 \subseteq V_2 \text{ then } [\![l]\!]^\sharp \; V_1 \subseteq [\![l]\!]^\sharp \; V_2.$$

Some facts:

If $f : \mathbb{D}_1 \rightarrow D_2$ and $g : \mathbb{D}_2 : \mathbb{D}_3$ are monotone then the composition $g \circ f : \mathbb{D}_1 \rightarrow D_3$ is monotone.

Some facts:

If $f : \mathbb{D}_1 \longrightarrow D_2$ and $g : \mathbb{D}_2 : \mathbb{D}_3$ are monotone then the composition $g \circ f : \mathbb{D}_1 \longrightarrow D_3$ is monotone.

If $\mathbb{D}_2$ is a complete lattice then the set $[\mathbb{D}_1 \longrightarrow \mathbb{D}_2]$ of monotone functions $f : \mathbb{D}_1 \longrightarrow \mathbb{D}_2$ is a complete lattice,

where $f \sqsubseteq g$ iff $f(x) \sqsubseteq g(x)$ for all $x \in \mathbb{D}_1$.

For $F \subseteq [\mathbb{D}_1 \longrightarrow \mathbb{D}_2]$ we have

$$\bigsqcup F = f \text{ with } f(x) = \bigsqcup\{g(x) \mid g \in F\} \qquad .$$

For our program analysis problem, we want the least solution of the constraint system

$$\begin{array}{ll} \mathcal{V}[0] \supseteq \mathcal{S} & (0 \text{ is the } \textit{start} \text{ node}) \\ \mathcal{V}[v] \supseteq [\![l]\!]^\sharp\ \mathcal{V}[u] & \text{for every edge } (u, l, v). \end{array}$$

We have the domain $\mathbb{D} = 2^{\mathcal{S}}$. Choose a variable for each set $\mathcal{V}[v]$.

We have a constraint system of the form

$$x_i \sqsupseteq f_i(x_1, \ldots, x_n) \qquad (1 \leq i \leq n)$$

Since $\mathbb{D}$ is a lattice, $\mathbb{D}^n$ is also a lattice where

$$(d_1, \ldots, d_n) \sqsubseteq (d'_1, \ldots, d'_n) \text{ iff } d_i \sqsubseteq d'_i \text{ for } 1 \leq i \leq n$$

The functions $f_i : \mathbb{D}^n \to \mathbb{D}$ are monotone.

Define $F : \mathbb{D}^n \longrightarrow \mathbb{D}^n$ as

$$F(y) = (f_1(y), \ldots, f_n(y)) \text{ where } y = (x_1, \ldots, x_n)$$

F is also monotone.

We need least solution of $y \sqsupseteq F(y)$.

Idea: use iteration

Start with the least element $\perp$ and compute the sequence

$\perp, F(\perp), F^2(\perp), F^3(\perp), \ldots.$

Do we always reach the least solution in this way?

Example: the complete lattice of Booleans: $\mathbb{D} = \{\bot, \top\}$.

Constraint system:

$$
\begin{aligned}
x &\sqsupseteq y \vee z \\
y &\sqsupseteq x \wedge y \wedge z \\
z &\sqsupseteq \top
\end{aligned}
$$

The iteration:

x	$\bot$			
y	$\bot$			
z	$\bot$			

We have $F^2(\bot) = F^3(\bot)$.

Example: the complete lattice of Booleans: $\mathbb{D} = \{\bot, \top\}$.

Constraint system:

$$
\begin{aligned}
x &\sqsupseteq y \vee z \\
y &\sqsupseteq x \wedge y \wedge z \\
z &\sqsupseteq \top
\end{aligned}
$$

The iteration:

x	$\bot$	$\bot$		
y	$\bot$	$\bot$		
z	$\bot$	$\top$		

We have $F^2(\bot) = F^3(\bot)$.

Example: the complete lattice of Booleans: $\mathbb{D} = \{\bot, \top\}$.

Constraint system:

$$\begin{aligned} x &\sqsupseteq y \vee z \\ y &\sqsupseteq x \wedge y \wedge z \\ z &\sqsupseteq \top \end{aligned}$$

The iteration:

x	$\bot$	$\bot$	$\top$	
y	$\bot$	$\bot$	$\bot$	
z	$\bot$	$\top$	$\top$	

We have $F^2(\bot) = F^3(\bot)$.

Example: the complete lattice of Booleans: $\mathbb{D} = \{\bot, \top\}$.

Constraint system:

$$\begin{aligned} x &\sqsupseteq y \vee z \\ y &\sqsupseteq x \wedge y \wedge z \\ z &\sqsupseteq \top \end{aligned}$$

The iteration:

x	$\bot$	$\bot$	$\top$	$\top$
y	$\bot$	$\bot$	$\bot$	$\bot$
z	$\bot$	$\top$	$\top$	$\top$

We have $F^2(\bot) = F^3(\bot)$.

Such an iteration produces an ascending chain

$$\perp \sqsubseteq F(\perp) \sqsubseteq F^2(\perp) \sqsubseteq F^3(\perp) \ldots$$

By induction: (1) Clearly $\perp \sqsubseteq F(\perp)$.

(2) Further if $F^i(\perp) \sqsubseteq F^{i+1}(\perp)$ then by monotonicity $F^{i+1}(\perp) \sqsubseteq F^{i+2}(\perp)$

Such an iteration produces an ascending chain

$$\bot \sqsubseteq F(\bot) \sqsubseteq F^2(\bot) \sqsubseteq F^3(\bot) \ldots$$

By induction: (1) Clearly $\bot \sqsubseteq F(\bot)$.

(2) Further if $F^i(\bot) \sqsubseteq F^{i+1}(\bot)$ then by monotonicity $F^{i+1}(\bot) \sqsubseteq F^{i+2}(\bot)$

Further if $F^k(\bot) = F^{k+1}(\bot)$ for some k
then clearly $F^k(\bot)$ is some solution of the constraint $F(x) \sqsubseteq x$.

Such an iteration produces an ascending chain

$$\bot \sqsubseteq F(\bot) \sqsubseteq F^2(\bot) \sqsubseteq F^3(\bot) \ldots$$

By induction: (1) Clearly $\bot \sqsubseteq F(\bot)$.

(2) Further if $F^i(\bot) \sqsubseteq F^{i+1}(\bot)$ then by monotonicity $F^{i+1}(\bot) \sqsubseteq F^{i+2}(\bot)$

Further if $F^k(\bot) = F^{k+1}(\bot)$ for some k
then clearly $F^k(\bot)$ is some solution of the constraint $F(x) \sqsubseteq x$.

Is it also the least solution of $F(x) \sqsubseteq x$?

Such an iteration produces an ascending chain

$$\perp \sqsubseteq F(\perp) \sqsubseteq F^2(\perp) \sqsubseteq F^3(\perp) \ldots$$

By induction: (1) Clearly $\perp \sqsubseteq F(\perp)$.

(2) Further if $F^i(\perp) \sqsubseteq F^{i+1}(\perp)$ then by monotonicity $F^{i+1}(\perp) \sqsubseteq F^{i+2}(\perp)$

Further if $F^k(\perp) = F^{k+1}(\perp)$ for some k
then clearly $F^k(\perp)$ is some solution of the constraint $F(x) \sqsubseteq x$.

Is it also the least solution of $F(x) \sqsubseteq x$?

Yes ...

Claim: If a is a solution of $F(x) \sqsubseteq x$ then $F^k(\bot) \sqsubseteq a$ for all k.

By induction: Clearly $\bot \sqsubseteq a$

Further if $F^k(\bot) \sqsubseteq a$ then by monotonicity we have $F^{k+1}(\bot) \sqsubseteq F(a) \sqsubseteq a$.

Claim: If a is a solution of $F(x) \sqsubseteq x$ then $F^k(\perp) \sqsubseteq a$ for all k.

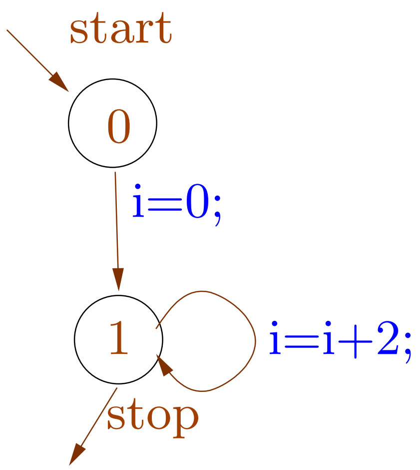
By induction: Clearly $\perp \sqsubseteq a$

Further if $F^k(\perp) \sqsubseteq a$ then by monotonicity we have $F^{k+1}(\perp) \sqsubseteq F(a) \sqsubseteq a$.

Hence if $F^{k+1}(\perp) = F^k(\perp)$ for any k then $F^k(\perp)$ is least solution of $F(x) \sqsubseteq x$.

Such a k always exists if the lattice is finite.

What in case of infinite lattices?



Constraint system:

$$\mathcal{V}[0] \supseteq \mathbb{Z}$$

$$\mathcal{V}[1] \supseteq \{0\} \cup \{x+2 \mid x \in \mathcal{V}[1]\}$$

The least solution:

$$\mathcal{V}[0] = \mathbb{Z} \text{ and } \mathcal{V}[1] = \{2n \mid n \geq 0\}.$$

Iteration doesn't terminate:

	$\bot$	$F(\bot)$	$F^2(\bot)$	$F^3(\bot)$	
$\mathcal{V}[0]$	$\emptyset$	$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$	$\ldots$
$\mathcal{V}[1]$	$\emptyset$	$\{0\}$	$\{0,2\}$	$\{0,2,4\}$	

Existence of least solutions: Knaster-Tarski

Fact: In a complete lattice $\mathbb{D}$, every monotone function $f : \mathbb{D} \to \mathbb{D}$ has a least fixpoint a.

Fixpoint: an element x such that $f(x) = x$.

Prefixpoint: an element x such that $f(x) \sqsubseteq x$.

Existence of least solutions: Knaster-Tarski

Fact: In a complete lattice $\mathbb{D}$, every monotone function $f : \mathbb{D} \to \mathbb{D}$ has a least fixpoint a.

> Fixpoint: an element x such that $f(x) = x$.
>
> Prefixpoint: an element x such that $f(x) \sqsubseteq x$.

Let $P = \{x \in \mathbb{D} \mid f(x) \sqsubseteq x\}$ (the set of prefixpoints).

The least fixpoint of f is $a = \bigsqcap P$.

(1) $a \in P$:

$$
\begin{array}{ll}
 & f(a) \sqsubseteq f(d) \sqsubseteq d \text{ for all } d \in P. \\
\Longrightarrow & f(a) \text{ is a lower bound of } P. \\
\Longrightarrow & f(a) \sqsubseteq a.
\end{array}
$$

$\Longrightarrow$ a is the least prefixpoint.

(2) $f(a) = a$:

$f(a) \sqsubseteq a$, from (1)

$\Longrightarrow$ $f^2(a) \sqsubseteq f(a)$, by monotonicity

$\Longrightarrow$ $f(a) \in P$

$\Longrightarrow$ $a \sqsubseteq f(a)$

Hence a is the least prefixpoint and is also a fixpoint.

Hence a is also the least fixpoint.

Example 1: Consider partial order $\mathbb{D}_1 = \mathbb{N}$ with $0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq \ldots$.

The function $f(x) = x{+}1$ is monotonic.

However it has no fixpoint.

Actually $\mathbb{D}_1$ is not a complete lattice.

Example 1: Consider partial order $\mathbb{D}_1 = \mathbb{N}$ with $0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq \ldots$.

The function $f(x) = x{+}1$ is monotonic.

However it has no fixpoint.

Actually $\mathbb{D}_1$ is not a complete lattice.

Example 2: Now we consider $\mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$.

This is a complete lattice.

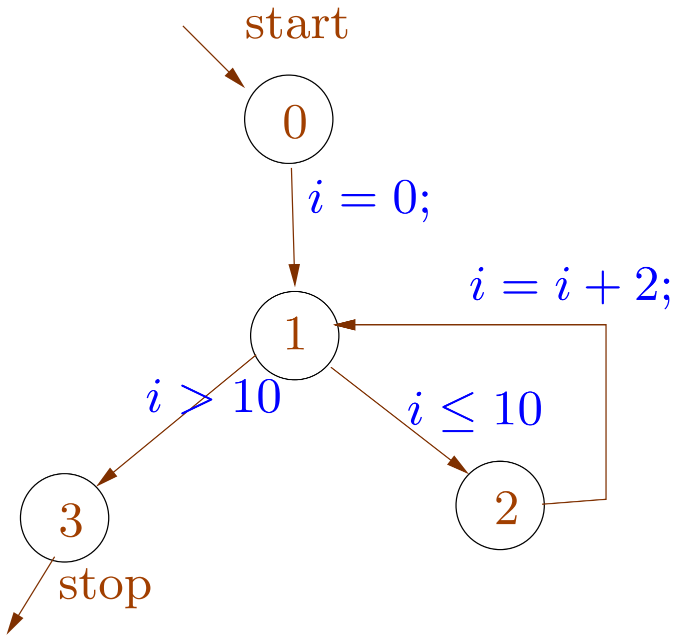
The function $f(x) = x{+}1$ is again monotonic.

The only fixpoint is ∞: $\infty{+}1 = \infty$.

Abstract Interpretation: Cousot, Cousot 1977

We use a suitable complete lattice as the domain of abstract values.

Example: intervals as abstract values:



$\mathcal{I}[0]$	$\bot$	$[-\infty, \infty]$	$[-\infty, \infty]$	$[-\infty, \infty]$	$[-\infty, \infty]$
$\mathcal{I}[1]$	$\bot$	$[0, 0]$	$[0, 2]$	$[0, 12]$	$[0, 12]$
$\mathcal{I}[2]$	$\bot$	$[0, 0]$	$[0, 2] \dots$	$[0, 10]$	$[0, 10]$
$\mathcal{I}[3]$	$\bot$	$\bot$	$\bot$	$[12, 12]$	$[12, 12]$

The analysis guarantees e.g. that at node 1 the value of i is always in the interval $[0, 12]$.

We have the set of concrete states $\mathcal{S} = (\mathsf{Vars} \to \mathbb{Z})$.

We choose a complete lattice $\mathbb{D}$ of abstract states.

We define an abstraction relation

$$\Delta \quad : \mathcal{S} \times \mathbb{D}$$

with the condition that

$$\rho \;\; \Delta \;\; a \quad \wedge \quad a \sqsubseteq b \quad \Longrightarrow \quad \rho \;\; \Delta \;\; b$$

$$\begin{tikzcd}
a \arrow[rr, "\sqsubseteq"] & & b \\
\rho \arrow[u, "\Delta"] \arrow[urr, dashed, "\Delta"'] & &
\end{tikzcd}$$

The concretization function: $\qquad \gamma(a) = \{\rho \mid \rho \;\; \Delta \;\; a\}.$

Example: For a program on two integer variables, $\mathsf{Vars} = \{x, y\}$.

The concrete states are from the set $\mathcal{S} = (\mathsf{Vars} \to \mathbb{Z})$ (or equivalently $\mathbb{Z}^2$).

For interval analysis, we choose the complete lattice

$$\mathbb{D}_{\mathbb{I}} = (\mathsf{Vars} \to \mathbb{I})_{\perp} = (\mathsf{Vars} \to \mathbb{I}) \cup \{\perp\}$$

where $\mathbb{I} = \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{\infty\}, l \leq u\}$ is the set of intervals.

Partial order on $\mathbb{I}$: $[l_1, u_1] \sqsubseteq [l_2, u_2]$ iff $l_1 \geq l_2$ and $u_1 \leq u_2$

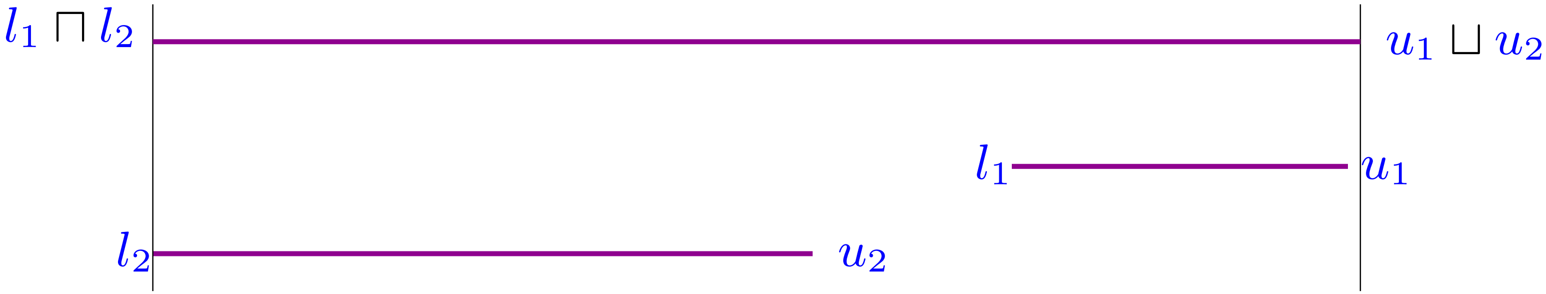
(As usual, $-\infty \sqsubseteq n \sqsubseteq \infty$ for all $n \in \mathbb{Z}$.)

Partial order on $\mathsf{Vars} \to \mathbb{I}$: $D_1 \sqsubseteq D_2$ iff $D_1(x) \sqsubseteq D_2(x)$.

Extension to $(\mathsf{Vars} \to \mathbb{I})_\perp$: $\perp \sqsubseteq D$ for all D.

$(\mathsf{Vars} \to \mathbb{I})_\perp$ is a complete lattice. $(\mathsf{Vars} \to \mathbb{I})$ is not.

In particular we define $[l_1, u_1] \sqcup [l_2, u_2] = [l_1 \sqcap l_2, u_1 \sqcup u_2]$.



$\perp$ represents the “unreachable state”: maps every variable to the “empty interval”.

The abstraction relation:

$$\rho \;\;\Delta\;\; D \qquad \text{iff} \qquad D \neq \bot \;\text{ and }\; \rho(x) \;\;\Delta\;\; D(x).$$

where $n \;\;\Delta\;\; [l, u]$ iff $l \leq n \leq u$.

The abstraction relation:

$$\rho \;\; \Delta \;\; D \quad \text{iff} \quad D \neq \bot \;\; \text{and} \;\; \rho(x) \;\; \Delta \;\; D(x).$$

where $n \;\; \Delta \;\; [l, u]$ iff $l \leq n \leq u$.

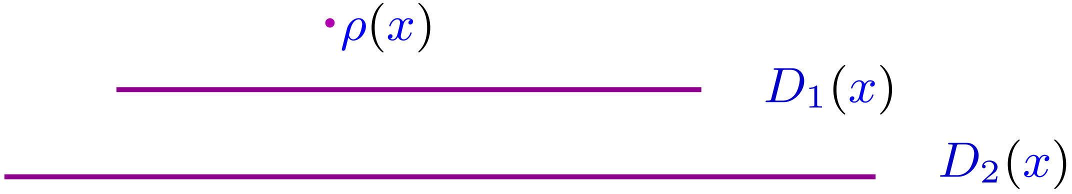
This satisfies the required condition:

Suppose $\rho \;\; \Delta \;\; D_1$ and $D_1 \sqsubseteq D_2$.

$\Longrightarrow$ $D_1 \neq \bot$ and $D_2 \neq \bot$.

$\rho(x) \;\; \Delta \;\; D_1(x)$ and $D_1(x) \sqsubseteq D_2(x)$ for each x.

$\Longrightarrow$ $\rho(x) \;\; \Delta \;\; D_1(x)$ for each x.



$\rho(x)$

$D_1(x)$

$D_2(x)$

The concretization function:

$$\gamma(\bot) = \{\}$$

$$\gamma(D) = \{\rho \mid \rho(x) \;\; \Delta \;\; D(x)\}, \qquad \text{for } D \neq \bot$$

$$\begin{aligned}\gamma(\{x \mapsto [3,5], y \mapsto [0,7]\}) = \qquad & \{\{x \mapsto 3, y \mapsto 0\}, \{x \mapsto 3, y \mapsto 1\}, \\ & \ldots \{x \mapsto 3, y \mapsto 7\} \\ & \ldots \{x \mapsto 5, y \mapsto 0\} \ldots \{x \mapsto 5, y \mapsto 7\}\}\end{aligned}$$

Abstraction of the partial transformation induced by edges.

Recall the edges $k = (u, l, v)$ induce a partial transformation on concrete states:

$$[\![k]\!] \;=[\![l]\!] \;: \mathcal{S} \rightarrow \mathcal{S}$$

Now on our chosen domain $\mathbb{D}$ we define a monotonic abstract transformation:

$$[\![k]\!]^\sharp \;= [\![l]\!]^\sharp \;: \mathbb{D} \rightarrow \mathbb{D}$$

The abstract transformation should simulate the concrete transformation:

$$\text{if} \quad \rho \;\Delta\; a \quad \text{and} \quad [\![l]\!]\,\rho \text{ is defined} \quad \text{then} \quad [\![l]\!]\,\rho \;\Delta\; [\![l]\!]^\sharp\, a.$$

$$\begin{tikzcd} a \arrow[r, "{[\![k]\!]^\sharp}"] & {} \\ \rho \arrow[u, "\Delta"] \arrow[r, "{[\![k]\!]}"'] & {} \arrow[u, dashed, "\Delta"'] \end{tikzcd}$$

Abstract transformation for interval analysis.

For concrete operators $\Box$ we define monotonic abstract operators $\Box^\sharp$ such that

$$x_1 \;\; \Delta \;\; a_1 \wedge \ldots \wedge x_n \;\; \Delta \;\; a_n \Longrightarrow \Box(x_1, \ldots, x_n) \quad \Delta \quad \Box^\sharp(a_1, \ldots, a_n)$$

addition:

$$\begin{aligned} [l_1, u_1] \quad +^\sharp \quad & [l_2, u_2] && = [l_1 + l_2, u_1 + u_2]. \\ \_ \quad + \quad & \infty && = \infty \\ \_ \quad + \quad & -\infty && = \infty \\ & && // \; \infty + -\infty \text{ is undefined.} \end{aligned}$$

substraction:

$$-^\sharp \quad [l, u] \qquad = [-u, -l]$$

Multiplication: $[l_1, u_1] \quad *^\sharp \quad [l_2, u_2] \quad = [m, n] \qquad$ where

$$m = l_1 l_2 \sqcap l_1 u_2 \sqcap l_2 u_1 \sqcap l_2 u_2$$

$$n = l_1 l_2 \sqcup l_1 u_2 \sqcup l_2 u_1 \sqcup l_2 u_2$$

Example:

$$\begin{aligned}
[1,3] \quad &*^\sharp \quad [5,8] &&= [5,24] \\
[-1,3] \quad &*^\sharp \quad [5,8] &&= [-8,24] \\
[-1,3] \quad &*^\sharp \quad [-5,8] &&= [-15,24] \\
[-1,3] \quad &*^\sharp \quad [-5,-8] &&= [-24,5]
\end{aligned}$$

Equality test:

$$[l_1, u_1] ==^\sharp [l_2, u_2] \;=\; \begin{cases} [1, 1] & \text{if} \quad l_1 = u_1 = l_2 = u_2 \\ [0, 0] & \text{if} \quad u_1 < l_2 \text{ or } u_2 < l_1 \\ [0, 1] & \text{otherwise} \end{cases}$$

Example:

$$\begin{array}{lcll} [7,7] & ==^\sharp & [7,7] & = [1,1] \\ [1,7] & ==^\sharp & [9,12] & = [0,0] \\ [1,7] & ==^\sharp & [1,7] & = [0,1] \end{array}$$

Inequality test:

$$[l_1, u_1] <^\sharp [l_2, u_2] \;=\; \begin{cases} [1,1] & \text{if} \quad u_1 < l_2 \\ [0,0] & \text{if} \quad u_2 < l_1 \\ [0,1] & \text{otherwise} \end{cases}$$

Example:

$$\begin{aligned} [1,7] &\quad <^\sharp \quad [9,12] &&= [1,1] \\ [9,12] &\quad <^\sharp \quad [1,7] &&= [0,0] \\ [1,7] &\quad <^\sharp \quad [6,8] &&= [0,1] \end{aligned}$$

Monotonic abstract evaluation of expressions

For $D \neq \bot$,

$$\begin{aligned}
[\![x]\!]^\sharp\ D &= D(x) \\
[\![n]\!]^\sharp\ D &= [n, n] \\
[\![\Box(e_1, \ldots, e_n)]\!]^\sharp\ D &= \Box^\sharp([\![e_1]\!]^\sharp\ D, \ldots, [\![e_n]\!]^\sharp\ D)
\end{aligned}$$

Monotonic abstract evaluation of expressions

For $D \neq \bot$,

$$\begin{aligned} [\![x]\!]^\sharp\ D &= D(x) \\ [\![n]\!]^\sharp\ D &= [n,n] \\ [\![\Box(e_1,\ldots,e_n)]\!]^\sharp\ D &= \Box^\sharp([\![e_1]\!]^\sharp\ D,\ldots,[\![e_n]\!]^\sharp\ D) \end{aligned}$$

Fact: $\rho \ \Delta \ D$ and $[\![e]\!]\ \rho$ is defined $\Longrightarrow [\![e]\!]\ \rho \ \Delta \ [\![e]\!]^\sharp\ D$.

Monotonic abstract evaluation of expressions

For $D \neq \bot$,

$$\begin{aligned} [\![x]\!]^\sharp\ D &= D(x) \\ [\![n]\!]^\sharp\ D &= [n, n] \\ [\![\Box(e_1, \ldots, e_n)]\!]^\sharp\ D &= \Box^\sharp([\![e_1]\!]^\sharp\ D, \ldots, [\![e_n]\!]^\sharp\ D) \end{aligned}$$

Fact: $\rho\ \ \Delta\ \ D$ and $[\![e]\!]\ \rho$ is defined $\Longrightarrow [\![e]\!]\ \rho\ \ \Delta\ \ [\![e]\!]^\sharp\ D$.

Case e is x: since $\rho\ \ \Delta\ \ D$ hence $[\![x]\!]\ \rho = \rho(x)\ \ \Delta\ \ D(x) = [\![x]\!]^\sharp\ D$

Monotonic abstract evaluation of expressions

For $D \neq \bot$,
$$\begin{aligned} [\![x]\!]^\sharp\ D &= D(x) \\ [\![n]\!]^\sharp\ D &= [n, n] \\ [\![\Box(e_1, \ldots, e_n)]\!]^\sharp\ D &= \Box^\sharp([\![e_1]\!]^\sharp\ D, \ldots, [\![e_n]\!]^\sharp\ D) \end{aligned}$$

Fact: $\rho\ \ \Delta\ \ D$ and $[\![e]\!]\ \rho$ is defined $\Longrightarrow [\![e]\!]\ \rho\ \ \Delta\ \ [\![e]\!]^\sharp\ D$.

Case e is x: since $\rho\ \ \Delta\ \ D$ hence $[\![x]\!]\ \rho = \rho(x)\ \ \Delta\ \ D(x) = [\![x]\!]^\sharp\ D$

Case e is n: $[\![n]\!]\ \rho = n\ \ \Delta\ \ [n, n] = [\![n]\!]^\sharp\ D$

Monotonic abstract evaluation of expressions

For $D \neq \bot$,

$$\begin{aligned} [\![x]\!]^\sharp\ D &= D(x) \\ [\![n]\!]^\sharp\ D &= [n, n] \\ [\![\Box(e_1, \ldots, e_n)]\!]^\sharp\ D &= \Box^\sharp([\![e_1]\!]^\sharp\ D, \ldots, [\![e_n]\!]^\sharp\ D) \end{aligned}$$

Fact: $\rho \ \Delta \ D$ and $[\![e]\!]\ \rho$ is defined $\Longrightarrow [\![e]\!]\ \rho \ \Delta \ [\![e]\!]^\sharp\ D$.

Case e is x: since $\rho \ \Delta \ D$ hence $[\![x]\!]\ \rho = \rho(x) \ \Delta \ D(x) = [\![x]\!]^\sharp\ D$

Case e is n: $[\![n]\!]\ \rho = n \ \Delta \ [n, n] = [\![n]\!]^\sharp\ D$

Case e is $\Box(e_1, \ldots, e_n)$: since each $[\![e_i]\!]\ \rho \ \Delta \ [\![e_i]\!]^\sharp\ D$ hence

$$\begin{aligned} [\![\Box(e_1, \ldots, e_n)]\!]\ \rho &= \Box([\![e_1]\!]\ \rho, \ldots, [\![e_n]\!]\ \rho) \\ &\Delta \\ \Box^\sharp([\![e_1]\!]^\sharp\ D, \ldots, [\![e_n]\!]^\sharp\ D) &= [\![\Box^\sharp(e_1, \ldots, e_n)]\!]^\sharp\ D \end{aligned}$$

Finally, the monotonic abstract transformations induced by edges

$$\begin{array}{rrl} & [\![l]\!]^\sharp\ \bot & = \bot \\ \text{For } D \neq \bot, & [\![;]\!]^\sharp\ D & = D \\ & [\![x = e;]\!]^\sharp\ D & = D \oplus \{x \mapsto [\![e]\!]^\sharp\ D\} \\ & [\![e]\!]^\sharp\ D & = \begin{cases} \bot & \text{if } [\![e]\!]^\sharp\ D = [0,0] \\ D & \text{otherwise} \end{cases} \end{array}$$

Finally, the monotonic abstract transformations induced by edges

$$\begin{array}{rll} & [\![l]\!]^\sharp \ \bot & = \bot \\ \text{For } D \neq \bot, & [\![;]\!]^\sharp \ D & = D \\ & [\![x = e;]\!]^\sharp \ D & = D \oplus \{x \mapsto [\![e]\!]^\sharp \ D\} \\ & [\![e]\!]^\sharp \ D & = \begin{cases} \bot & \text{if } [\![e]\!]^\sharp \ D = [0, 0] \\ D & \text{otherwise} \end{cases} \end{array}$$

Next we must check the condition:

$$\rho \ \ \Delta \ \ D \quad \wedge \quad [\![l]\!] \ \rho = \rho_1 \quad \wedge \quad [\![l]\!]^\sharp \ D = D_1 \quad \Longrightarrow \quad \rho_1 \ \ \Delta \ \ D_1 .$$

Finally, the monotonic abstract transformations induced by edges

$$\begin{array}{rrl} & [\![l]\!]^\sharp\ \bot & = \bot \\ \text{For } D \neq \bot, & [\![;]\!]^\sharp\ D & = D \\ & [\![x = e;]\!]^\sharp\ D & = D \oplus \{x \mapsto [\![e]\!]^\sharp\ D\} \\ & [\![e]\!]^\sharp\ D & = \begin{cases} \bot & \text{if } [\![e]\!]^\sharp\ D = [0,0] \\ D & \text{otherwise} \end{cases} \end{array}$$

Next we must check the condition:

$$\rho \ \Delta \ D \quad \wedge \quad [\![l]\!]\ \rho = \rho_1 \quad \wedge \quad [\![l]\!]^\sharp\ D = D_1 \quad \Longrightarrow \quad \rho_1 \ \Delta \ D_1.$$

Clearly $D \neq \bot$ here.

To check: $\rho \;\; \Delta \;\; D \;\; \wedge \;\; [\![l]\!] \, \rho = \rho_1 \;\; \wedge \;\; [\![l]\!]^\sharp \, D = D_1 \;\; \Longrightarrow \;\; \rho_1 \;\; \Delta \;\; D_1$.

Case l is ;

$$\rho_1 = \rho \;\; \Delta \;\; D = D_1.$$

To check: $\quad \rho \;\; \Delta \;\; D \;\; \wedge \;\; [\![l]\!]\, \rho = \rho_1 \;\; \wedge \;\; [\![l]\!]^\sharp\, D = D_1 \;\; \Longrightarrow \;\; \rho_1 \;\; \Delta \;\; D_1.$

Case l is ;

$$\rho_1 = \rho \;\; \Delta \;\; D = D_1.$$

Case l is $x = e$;

$$\rho_1 = \rho \oplus \{x \mapsto [\![e]\!]\, \rho\} \quad \text{and} \quad D_1 = D \oplus \{x \mapsto [\![e]\!]^\sharp\, D\}$$

As $\;[\![e]\!]\, \rho \;\; \Delta \;\; [\![e]\!]^\sharp\, D\;$ hence $\;\rho_1 \;\; \Delta \;\; D_1.$

To check: $\rho \;\; \Delta \;\; D \quad \wedge \quad [\![l]\!]\, \rho = \rho_1 \quad \wedge \quad [\![l]\!]^\sharp\, D = D_1 \quad \Longrightarrow \quad \rho_1 \;\; \Delta \;\; D_1.$

Case l is ;

$$\rho_1 = \rho \;\; \Delta \;\; D = D_1.$$

Case l is $x = e$;

$$\rho_1 = \rho \oplus \{x \mapsto [\![e]\!]\; \rho\} \quad \text{and} \quad D_1 = D \oplus \{x \mapsto [\![e]\!]^\sharp\; D\}$$

As $[\![e]\!]\; \rho \;\; \Delta \;\; [\![e]\!]^\sharp\; D$ hence $\rho_1 \;\; \Delta \;\; D_1$.

Case e is some condition e

Since the tranformation $[\![e]\!]\; \rho$ is defined,

hence the expression evaluation $[\![e]\!]\; \rho \neq 0$, and $\rho_1 = \rho$.

Since $\rho \;\; \Delta \;\; D$,

hence the abstract expression evaluation $[\![e]\!]^\sharp\; D \neq [0, 0]$, and $D_1 = D$.

Recall, for a path $\pi = k_1 \dots k_n$,

$$\llbracket \pi \rrbracket \ \rho = (\llbracket k_n \rrbracket \ \circ \dots \circ \llbracket k_1 \rrbracket \ )\rho$$

$$\llbracket \pi \rrbracket^\sharp \ D = (\llbracket k_n \rrbracket^\sharp \ \circ \dots \circ \llbracket k_1 \rrbracket^\sharp \ )D$$

We conclude from above:

if $\rho \ \ \Delta \ \ D$ and $\llbracket \pi \rrbracket \ \rho$ is defined then $\llbracket \pi \rrbracket \ \rho \ \ \Delta \ \ \llbracket \pi \rrbracket^\sharp \ D$.

$$\begin{tikzcd}
D \arrow[r, "\llbracket k_1 \rrbracket^\sharp"] & {} \arrow[r, "\llbracket k_2 \rrbracket^\sharp"] & {} \arrow[r, "\llbracket k_3 \rrbracket^\sharp"] & {} & \cdots & {} \arrow[r, "\llbracket k_n \rrbracket^\sharp"] & {} \\
\rho \arrow[u, "\Delta"] \arrow[r, "\llbracket k_1 \rrbracket"'] & {} \arrow[u, dashed, "\Delta"] \arrow[r, "\llbracket k_2 \rrbracket"'] & {} \arrow[u, dashed, "\Delta"] \arrow[r, "\llbracket k_3 \rrbracket"'] & {} \arrow[u, dashed, "\Delta"] & & {} \arrow[u, dashed, "\Delta"] \arrow[r, "\llbracket k_n \rrbracket"'] & {} \arrow[u, dashed, "\Delta"]
\end{tikzcd}$$

Merge over All Paths (MOP):

$$\mathcal{D}^*[v] = \bigsqcup \{ [\![\pi]\!]^\sharp \; \top \mid \pi : \mathit{start} \rightarrow^* v \}$$

For any initial concrete state ρ and path $\pi : \mathit{start} \rightarrow^* v$, if $[\![\pi]\!] \; \rho$ is defined then

$$[\![\pi]\!] \; \rho \quad \Delta \quad \mathcal{D}^*[v]$$

Hence $\mathcal{D}^*[v]$ abstracts all states possible at node v.

Merge over All Paths (MOP):

$$\mathcal{D}^*[v] = \bigsqcup\{[\![\pi]\!]^\sharp\ \top \mid \pi : \mathit{start} \to^* v\}$$

For any initial concrete state ρ and path $\pi : \mathit{start} \to^* v$, if $[\![\pi]\!]\ \rho$ is defined then

$$[\![\pi]\!]\ \rho \quad \Delta \quad \mathcal{D}^*[v]$$

Hence $\mathcal{D}^*[v]$ abstracts all states possible at node v.

To compute it, we use the constraint system $\mathcal{D}^*$.

$$\begin{array}{lll} \mathcal{D}[\mathit{start}] & \sqsupseteq \top & \\ \mathcal{D}[v] & \sqsupseteq [\![k]\!]^\sharp\ \mathcal{D}[u] & \text{for edge } k = (u, l, v) \end{array}$$

Merge over All Paths (MOP):

$$\mathcal{D}^*[v] = \bigsqcup\{[\![\pi]\!]^\sharp\ \top \mid \pi : start \to^* v\}$$

For any initial concrete state ρ and path $\pi : start \to^* v$, if $[\![\pi]\!]\ \rho$ is defined then

$$[\![\pi]\!]\ \rho \quad \Delta \quad \mathcal{D}^*[v]$$

Hence $\mathcal{D}^*[v]$ abstracts all states possible at node v.

To compute it, we use the constraint system $\mathcal{D}^*$.

$$\begin{array}{lll} \mathcal{D}[start] & \sqsupseteq \top & \\ \mathcal{D}[v] & \sqsupseteq [\![k]\!]^\sharp\ \mathcal{D}[u] & \text{for edge } k = (u, l, v) \end{array}$$

How are the two related?

Merge over All Paths (MOP):

$$\mathcal{D}^*[v] = \bigsqcup\{[\![\pi]\!]^\sharp\ D_0 \mid \pi : \mathit{start} \to^* v\}$$

Theorem: Kam,Ullman 1975

Let $\mathcal{D}$ be the smallest solution of the constraint system

$$\begin{array}{lll} \mathcal{D}[\mathit{start}] & \sqsupseteq D_0 & \\ \mathcal{D}[v] & \sqsupseteq [\![k]\!]^\sharp\ \mathcal{D}[u] & \text{for edge } k = (u, l, v) \end{array}$$

Then we have

$$\begin{array}{rll} & \mathcal{D}[v] \sqsupseteq \mathcal{D}^*[v] & \text{for every } v \\ \text{In other words:} & \mathcal{D}[v] \sqsupseteq [\![\pi]\!]^\sharp\ D_0 & \text{for every } \pi : \mathit{start} \to^* v \end{array}$$

Proof: induction on the length of π:

Proof: induction on the length of π:

Case $\pi = \epsilon$ (empty path).

Proof: induction on the length of π:

Case $\pi = \epsilon$ (empty path).

$$[\![\pi]\!]^\sharp\ D_0 = D_0 \sqsubseteq \mathcal{D}[start]$$

Proof: induction on the length of π:

Case $\pi = \epsilon$ (empty path).

$$[\![\pi]\!]^\sharp\; D_0 = D_0 \sqsubseteq \mathcal{D}[start]$$

Induction step: $\pi = \pi' k$ for $k = (u, l, v)$.

Proof: induction on the length of π:

Case $\pi = \epsilon$ (empty path).

$$[\![\pi]\!]^\sharp\; D_0 = D_0 \sqsubseteq \mathcal{D}[start]$$

Induction step: $\pi = \pi' k$ for $k = (u, l, v)$.

$$\begin{array}{lll} [\![\pi']\!]^\sharp\; D_0 & \sqsubseteq \mathcal{D}[u] & \text{induction hypothesis} \\ [\![\pi]\!]^\sharp\; D_0 & = [\![k]\!]^\sharp\; ([\![\pi']\!]^\sharp\; D_0) & \\ & \sqsubseteq [\![k]\!]^\sharp\; (\mathcal{D}[u]) & \text{monotonicity} \\ & \sqsubseteq \mathcal{D}[v] & \mathcal{D} \text{ is a solution} \end{array}$$

Question:

Does the constraint system give us only an upper bound ?

Question:

Does the constraint system give us only an upper bound ?

Answer:

In general yes.

Question:

Does the constraint system give us only an upper bound ?

Answer:

In general yes.

Now let's assume that all the functions $[\![k]\!]^\sharp$ are distributive …

A function $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called

- distributive, when $f(\bigsqcup X) = \bigsqcup\{f(x) \mid x \in X\}$ for all $\emptyset \neq X \subseteq \mathbb{D}_1$.
- strict, when $f(\bot) = \bot$.
- total distributive, when f is strict and distributive.

A function $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called

- distributive, when $f(\bigsqcup X) = \bigsqcup\{f(x) \mid x \in X\}$ for all $\emptyset \neq X \subseteq \mathbb{D}_1$.
- strict, when $f(\bot) = \bot$.
- total distributive, when f is strict and distributive.

Example 1: $\mathbb{D}_1 = \mathbb{D}_2 = (2^U, \subseteq)$ for some set U.

$f(x) = x \cap A \cup B$ for some $A, B \subseteq U$.

A function $f : \mathbb{D}_1 \rightarrow \mathbb{D}_2$ is called

- distributive, when $f(\bigsqcup X) = \bigsqcup\{f(x) \mid x \in X\}$ for all $\emptyset \neq X \subseteq \mathbb{D}_1$.
- strict, when $f(\bot) = \bot$.
- total distributive, when f is strict and distributive.

Example 1: $\mathbb{D}_1 = \mathbb{D}_2 = (2^U, \subseteq)$ for some set U.

$f(x) = x \cap A \cup B$ for some $A, B \subseteq U$.

Strictness: $f(\emptyset) = B \implies$ strict only if $B = \emptyset$.

A function $f : \mathbb{D}_1 \to \mathbb{D}_2$ is called

- distributive, when $f(\bigsqcup X) = \bigsqcup\{f(x) \mid x \in X\}$ for all $\emptyset \neq X \subseteq \mathbb{D}_1$.
- strict, when $f(\bot) = \bot$.
- total distributive, when f is strict and distributive.

Example 1: $\mathbb{D}_1 = \mathbb{D}_2 = (2^U, \subseteq)$ for some set U.

$f(x) = x \cap A \cup B$ for some $A, B \subseteq U$.

Strictness: $f(\emptyset) = B \implies$ strict only if $B = \emptyset$.

Distributivity:

$$\begin{aligned} f(x \cup y) &= (x \cup y) \cap A \cup B \\ &= (x \cap A) \cup (y \cap A) \cup B \\ &= (x \cap A \cup B) \cup (y \cap A \cup B) \qquad \text{:-)} \end{aligned}$$

Example 2: $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \qquad f(x) = x+1.$

Example 2: $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \qquad f(x) = x+1.$

Strictness: $f(\bot) = 0+1 = 1 \neq \bot$:-(

Example 2: $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}$, $f(x) = x+1$.

Strictness: $f(\bot) = 0+1 = 1 \neq \bot$:-(

Distributivity: $f(\bigsqcup X) = 1 + \bigsqcup X = \bigsqcup\{x+1 \mid x \in X\} = \bigsqcup\{f(x) \mid x \in X\}$ for $\emptyset \neq X$:-)

Example 2: $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad f(x) = x+1.$

Strictness: $f(\bot) = 0+1 = 1 \neq \bot$:-(

Distributivity: $f(\bigsqcup X) = 1+\bigsqcup X = \bigsqcup\{x+1 \mid x \in X\} = \bigsqcup\{f(x) \mid x \in X\}$ for $\emptyset \neq X$:-)

Example 3: $\mathbb{D}_1 = (\mathbb{N} \cup \{\infty\})^2, \quad \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad f(x,y) = x+y$

Example 2: $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad f(x) = x+1.$

Strictness: $f(\bot) = 0+1 = 1 \neq \bot$:-(

Distributivity: $f(\bigsqcup X) = 1 + \bigsqcup X = \bigsqcup\{x+1 \mid x \in X\} = \bigsqcup\{f(x) \mid x \in X\}$ for $\emptyset \neq X$:-)

Example 3: $\mathbb{D}_1 = (\mathbb{N} \cup \{\infty\})^2, \quad \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad f(x,y) = x+y$

Strictness: $f(\bot) = 0+0 = 0 = \bot$:-)

Example 2: $\mathbb{D}_1 = \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad f(x) = x{+}1.$

Strictness: $f(\bot) = 0{+}1 = 1 \neq \bot$:-(

Distributivity: $f(\bigsqcup X) = 1{+}\bigsqcup X = \bigsqcup\{x{+}1 \mid x \in X\} = \bigsqcup\{f(x) \mid x \in X\}$ for $\emptyset \neq X$:-)

Example 3: $\mathbb{D}_1 = (\mathbb{N} \cup \{\infty\})^2, \quad \mathbb{D}_2 = \mathbb{N} \cup \{\infty\}, \quad f(x, y) = x{+}y$

Strictness: $f(\bot) = 0{+}0 = 0 = \bot$:-)

Distributivity: $f((1,4) \sqcup (4,1)) = f(4,4) = 8 \neq 5 = f(1,4) \sqcup f(4,1)$:-(

Assumption: All nodes v are reachable from the node $start$.

(Unreachable nodes can always be deleted.)

Theorem: If all the edge transofrmations $[\![k]\!]^\sharp$ are distributive then $\mathcal{D}^*[v] = \mathcal{D}[v]$ for all v.

Assumption: All nodes v are reachable from the node $start$.

(Unreachable nodes can always be deleted.)

Theorem: If all the edge transofrmations $[\![k]\!]^\sharp$ are distributive then $\mathcal{D}^*[v] = \mathcal{D}[v]$ for all v.

Proof: We show that $\mathcal{D}^*$ satisfies the constraint system.

(1) For the *start* node:

$$\begin{aligned} \mathcal{D}^*[\mathit{start}] &= \bigsqcup\{[\![\pi]\!]^\sharp\ D_0 \mid \pi : \mathit{start} \to \mathit{start}\} \\ &\sqsupseteq [\![\epsilon]\!]^\sharp\ D_0 \\ &= D_0 \end{aligned}$$

(1) For the *start* node:

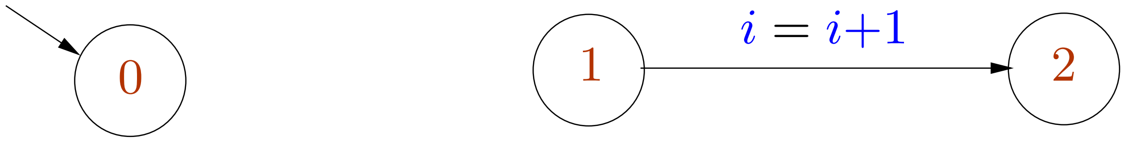
$$\begin{aligned}\mathcal{D}^*[start] &= \bigsqcup\{[\![\pi]\!]^\sharp\ D_0 \mid \pi : start \rightarrow start\} \\ &\sqsupseteq [\![\epsilon]\!]^\sharp\ D_0 \\ &= D_0\end{aligned}$$

(2) For every edge $k = (u, l, v)$

$$\begin{aligned}\mathcal{D}^*[v] &= \bigsqcup\{[\![\pi]\!]^\sharp\ D_0 \mid \pi : start \rightarrow v\} \\ &\sqsupseteq \bigsqcup\{[\![\pi' k]\!]^\sharp\ D_0 \mid \pi' : start \rightarrow u\} \\ &= \bigsqcup\{[\![k]\!]^\sharp\ ([\![\pi']\!]^\sharp\ D_0) \mid \pi' : start \rightarrow u\} \\ &= [\![k]\!]^\sharp\ (\bigsqcup\{[\![\pi']\!]^\sharp\ D_0 \mid \pi' : start \rightarrow u\}) \\ &= [\![k]\!]^\sharp\ (\mathcal{D}^*[u])\end{aligned}$$

since $\{\pi' \mid \pi' : start \rightarrow u\}$ is non-empty.

The result does not hold in case of unreachable nodes.



We consider $\mathbb{D} = \mathbb{N} \cup \{\infty\}$ with ordering $0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq \ldots \sqsubseteq \infty$.

Abstraction relation: $n \;\; \Delta \;\; a$ iff $n \leq a$.

The abstract transformation for the second edge is defined by $[\![k]\!]^\sharp\; a = a{+}1$.

We choose $D_0 = 5$.

We have the constraints $\mathcal{D}[0] \sqsupseteq 5$ and $\mathcal{D}[2] \sqsupseteq \mathcal{D}[1]{+}1$.

We have

$$\mathcal{D}^*[2] = \bigsqcup \emptyset = 0$$

$$\mathcal{D}[2] = 0{+}1 = 1$$

The Notion of Type Safety

Use typing rules to filter out unsafe programs.

Two kinds of semantics of programs:

The Notion of Type Safety

Use typing rules to filter out unsafe programs.

Two kinds of semantics of programs:

- Static semantics: types
- Dynamic semantics: execution of the program

The Notion of Type Safety

Use typing rules to filter out unsafe programs.

Two kinds of semantics of programs:

- Static semantics: types
- Dynamic semantics: execution of the program

Type safety: "Well types programs never go wrong"

– Robin Milner

Standard methodology: Safety = Progress + Preservation

Progress: a well types program that is not a value can be evaluated further

Preservation: well typed programs remain so during evaluation.

A simple functional language (the simply typed lambda calculus)

t	$::=$		terms:
		x	variable
	$\mid$	0	
	$\mid$	succ t $\mid$ pred t	
	$\mid$	iszero t	zero test
	$\mid$	true $\mid$ false	
	$\mid$	if t then t else t	
	$\mid$	fun $x : T \cdot t$	functions
	$\mid$	apply $(t\ , t\ )$	application

The types

$T ::=$

Bool	type of Booleans
Int	type of ints
$T \to T$	type of functions

The types

$T ::=$

	Bool	type of Booleans
	Int	type of ints
	$T \rightarrow T$	type of functions

The results of computations

$v ::=$ values:

	$\mathsf{true} \mid \mathsf{false}$	Boolean values
$\mid$	nv	numerical value
$\mid$	$\mathsf{fun}\ x : T \cdot t$	functional value

$nv ::=$

	0
$\mid$	$\mathsf{succ}\ nv$

The Dynamic Semantics: Evaluation

$$\frac{t \longrightarrow t'}{\mathsf{succ}\ t \longrightarrow \mathsf{succ}\ t'}\ \text{(E-Succ)}$$

The Dynamic Semantics: Evaluation

$$\frac{t \longrightarrow t'}{\mathsf{succ}\ t \longrightarrow \mathsf{succ}\ t'}\ \text{(E-Succ)} \qquad \frac{t \longrightarrow t'}{\mathsf{pred}\ t \longrightarrow \mathsf{pred}\ t'}\ \text{(E-Pred)}$$

The Dynamic Semantics: Evaluation

$$\frac{t \longrightarrow t'}{\textsf{succ}\ t \longrightarrow \textsf{succ}\ t'} \text{ (E-Succ)} \qquad \frac{t \longrightarrow t'}{\textsf{pred}\ t \longrightarrow \textsf{pred}\ t'} \text{ (E-Pred)}$$

$$\textsf{pred}\ 0 \longrightarrow 0 \text{ (E-PredZero)}$$

The Dynamic Semantics: Evaluation

$$\frac{t \longrightarrow t'}{\textsf{succ}\ t \longrightarrow \textsf{succ}\ t'}\ \text{(E-Succ)} \qquad \frac{t \longrightarrow t'}{\textsf{pred}\ t \longrightarrow \textsf{pred}\ t'}\ \text{(E-Pred)}$$

$$\textsf{pred}\ 0 \longrightarrow 0\ \text{(E-PredZero)} \qquad \textsf{pred}\ (\textsf{succ}\ nv) \longrightarrow nv\ \text{(E-PredSucc)}$$

The Dynamic Semantics: Evaluation

$$\frac{t \longrightarrow t'}{\textsf{succ}\ t \longrightarrow \textsf{succ}\ t'}\ \text{(E-Succ)} \qquad \frac{t \longrightarrow t'}{\textsf{pred}\ t \longrightarrow \textsf{pred}\ t'}\ \text{(E-Pred)}$$

$$\textsf{pred}\ 0 \longrightarrow 0\ \text{(E-PredZero)} \qquad \textsf{pred}\ (\textsf{succ}\ nv) \longrightarrow nv\ \text{(E-PredSucc)}$$

$$\frac{t \longrightarrow t'}{\textsf{iszero}\ t \longrightarrow \textsf{iszero}\ t'}\ \text{(E-IsZero)}$$

The Dynamic Semantics: Evaluation

$$\frac{t \longrightarrow t'}{\mathsf{succ}\ t \longrightarrow \mathsf{succ}\ t'} \text{ (E-Succ)} \qquad \frac{t \longrightarrow t'}{\mathsf{pred}\ t \longrightarrow \mathsf{pred}\ t'} \text{ (E-Pred)}$$

$$\mathsf{pred}\ 0 \longrightarrow 0 \text{ (E-PredZero)} \qquad \mathsf{pred}\ (\mathsf{succ}\ nv) \longrightarrow nv \text{ (E-PredSucc)}$$

$$\frac{t \longrightarrow t'}{\mathsf{iszero}\ t \longrightarrow \mathsf{iszero}\ t'} \text{ (E-IsZero)}$$

$$\mathsf{iszero}\ 0 \longrightarrow \mathsf{true} \quad \text{(E-IsZeroZero)}$$

The Dynamic Semantics: Evaluation

$$\frac{t \longrightarrow t'}{\mathsf{succ}\ t \longrightarrow \mathsf{succ}\ t'} \ \text{(E-Succ)} \qquad \frac{t \longrightarrow t'}{\mathsf{pred}\ t \longrightarrow \mathsf{pred}\ t'} \ \text{(E-Pred)}$$

$$\mathsf{pred}\ 0 \longrightarrow 0 \ \text{(E-PredZero)} \qquad \mathsf{pred}\ (\mathsf{succ}\ nv) \longrightarrow nv \ \text{(E-PredSucc)}$$

$$\frac{t \longrightarrow t'}{\mathsf{iszero}\ t \longrightarrow \mathsf{iszero}\ t'} \ \text{(E-IsZero)}$$

$$\mathsf{iszero}\ 0 \longrightarrow \mathsf{true} \quad \text{(E-IsZeroZero)} \qquad \mathsf{iszero}\ (\mathsf{succ}\ nv) \longrightarrow \mathsf{false} \quad \text{(E-IsZeroSucc)}$$

The Dynamic Semantics: Evaluation

$$\frac{t \longrightarrow t'}{\textsf{succ}\ t \longrightarrow \textsf{succ}\ t'}\ (\text{E-Succ}) \qquad \frac{t \longrightarrow t'}{\textsf{pred}\ t \longrightarrow \textsf{pred}\ t'}\ (\text{E-Pred})$$

$$\textsf{pred}\ 0 \longrightarrow 0\ (\text{E-PredZero}) \qquad \textsf{pred}\ (\textsf{succ}\ nv) \longrightarrow nv\ (\text{E-PredSucc})$$

$$\frac{t \longrightarrow t'}{\textsf{iszero}\ t \longrightarrow \textsf{iszero}\ t'}\ (\text{E-IsZero})$$

$$\textsf{iszero}\ 0 \longrightarrow \textsf{true}\quad (\text{E-IsZeroZero}) \qquad \textsf{iszero}\ (\textsf{succ}\ nv) \longrightarrow \textsf{false}\quad (\text{E-IsZeroSucc})$$

$$\frac{t \longrightarrow t'}{\textsf{if}\ t\ \textsf{then}\ t_1\ \textsf{else}\ t_2 \longrightarrow \textsf{if}\ t'\ \textsf{then}\ t_1\ \textsf{else}\ t_2}\ (\text{E-If})$$

$\mathsf{if\ true\ then}\ t_1\ \mathsf{else}\ t_2 \longrightarrow t_1$ (E-IfTrue)

$$\textsf{if true then } t_1 \textsf{ else } t_2 \longrightarrow t_1 \quad \text{(E-IfTrue)} \quad \textsf{if false then } t_1 \textsf{ else } t_2 \longrightarrow t_2 \quad \text{(E-IfFalse)}$$

$\textsf{if true then } t_1 \textsf{ else } t_2 \longrightarrow t_1$ (E-IfTrue) $\textsf{if false then } t_1 \textsf{ else } t_2 \longrightarrow t_2$ (E-IfFalse)

$$\frac{t_1 \longrightarrow t_1'}{\textsf{apply } (t_1\ , t_2\ ) \longrightarrow \textsf{apply } (t_1'\ , t_2\ )} \text{ (E-App1)}$$

$$\textsf{if true then } t_1 \textsf{ else } t_2 \longrightarrow t_1 \quad \text{(E-IfTrue)} \qquad \textsf{if false then } t_1 \textsf{ else } t_2 \longrightarrow t_2 \quad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\textsf{apply } (t_1, t_2) \longrightarrow \textsf{apply } (t_1', t_2)} \text{ (E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\textsf{apply } (v_1, t_2) \longrightarrow \textsf{apply } (v_1, t_2')} \text{ (E-App2)}$$

$$\textsf{if true then } t_1 \textsf{ else } t_2 \longrightarrow t_1 \quad \text{(E-IfTrue)} \qquad \textsf{if false then } t_1 \textsf{ else } t_2 \longrightarrow t_2 \quad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\textsf{apply } (t_1, t_2) \longrightarrow \textsf{apply } (t_1', t_2)} \text{ (E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\textsf{apply } (v_1, t_2) \longrightarrow \textsf{apply } (v_1, t_2')} \text{ (E-App2)}$$

$$\textsf{apply } (\textsf{fun } x : T \cdot t, v) \longrightarrow t\ [x \mapsto v] \text{ (E-App)}$$

$$\text{if true then } t_1 \text{ else } t_2 \longrightarrow t_1 \quad \text{(E-IfTrue)} \qquad \text{if false then } t_1 \text{ else } t_2 \longrightarrow t_2 \quad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{apply } (t_1, t_2) \longrightarrow \text{apply } (t_1', t_2)} \text{ (E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\text{apply } (v_1, t_2) \longrightarrow \text{apply } (v_1, t_2')} \text{ (E-App2)}$$

$$\text{apply } (\text{fun } x : T \cdot t, v) \longrightarrow t\,[x \mapsto v] \text{ (E-App)}$$

Substitutions are defined as usual.

$$(\text{if true then } (\text{pred } x) \text{ else } 0)\ [x \mapsto \text{succ } 0] = (\text{if true then } (\text{pred } (\text{succ } 0)) \text{ else } 0)$$

$$\textsf{if true then } t_1 \textsf{ else } t_2 \;\longrightarrow\; t_1 \quad \text{(E-IfTrue)} \qquad \textsf{if false then } t_1 \textsf{ else } t_2 \;\longrightarrow\; t_2 \quad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\textsf{apply } (t_1, t_2) \longrightarrow \textsf{apply } (t_1', t_2)} \text{ (E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\textsf{apply } (v_1, t_2) \longrightarrow \textsf{apply } (v_1, t_2')} \text{ (E-App2)}$$

$$\textsf{apply } (\textsf{fun } x : T \cdot t, v) \longrightarrow t\,[x \mapsto v] \text{ (E-App)}$$

Substitutions are defined as usual.

$$(\textsf{if true then } (\textsf{pred } x) \textsf{ else } 0)\,[x \mapsto \textsf{succ } 0] = (\textsf{if true then } (\textsf{pred } (\textsf{succ } 0)) \textsf{ else } 0)$$

$$(\textsf{fun } x : \textsf{Int} \cdot \textsf{if true then } x \textsf{ else succ } (y))[y \mapsto \textsf{succ } (x)]$$
$$= (\textsf{fun } z : \textsf{Int} \cdot \textsf{if true then } z \textsf{ else succ } (\textsf{succ } (x)))$$

Example

$$
\begin{array}{rl}
& \mathsf{apply}\ (\mathsf{fun}\ x : \mathsf{Int} \cdot \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pred}\ (\mathsf{succ}\ 0))\ \mathsf{else}\ (\mathsf{succ}\ 0), \mathsf{iszero}\ 0) \\
\longrightarrow & \mathsf{apply}\ (\mathsf{fun}\ x : \mathsf{Int} \cdot \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pred}\ (\mathsf{succ}\ 0))\ \mathsf{else}\ (\mathsf{succ}\ 0), \mathsf{true}\ ) \\
\longrightarrow & \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ (\mathsf{pred}\ (\mathsf{succ}\ 0))\ \mathsf{else}\ (\mathsf{succ}\ 0) \\
\longrightarrow & (\mathsf{pred}\ (\mathsf{succ}\ 0)) \\
\longrightarrow & 0
\end{array}
$$

Example

$$\begin{array}{rl}
 & \mathsf{apply}\ (\mathsf{fun}\ x : \mathsf{Int} \cdot \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pred}\ (\mathsf{succ}\ 0))\ \mathsf{else}\ (\mathsf{succ}\ 0), \mathsf{iszero}\ 0) \\
\longrightarrow & \mathsf{apply}\ (\mathsf{fun}\ x : \mathsf{Int} \cdot \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pred}\ (\mathsf{succ}\ 0))\ \mathsf{else}\ (\mathsf{succ}\ 0), \mathsf{true}\ ) \\
\longrightarrow & \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ (\mathsf{pred}\ (\mathsf{succ}\ 0))\ \mathsf{else}\ (\mathsf{succ}\ 0) \\
\longrightarrow & (\mathsf{pred}\ (\mathsf{succ}\ 0)) \\
\longrightarrow & 0
\end{array}$$

The justification for the first evaluation step is as follows

$$\frac{\dfrac{}{\mathsf{iszero}\ 0 \longrightarrow \mathsf{true}}\ (\text{E-IsZeroZero})}{\mathsf{apply}\ (\mathsf{fun}\ x : \mathsf{Int} \cdot \mathsf{if}\ \ \ldots, \mathsf{iszero}\ 0) \longrightarrow \mathsf{apply}\ (\mathsf{fun}\ x : \mathsf{Int} \cdot \mathsf{if}\ \ \ldots, \mathsf{true}\ )}\ (\text{E-App2})$$

A program which gets stuck during evaluation

$$\begin{array}{rl} & \mathsf{apply}\ (\mathsf{fun}\ x : \mathsf{Int} \cdot \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pred}\ (\mathsf{succ}\ 0))\ \mathsf{else}\ (\mathsf{succ}\ 0), 0) \\ \longrightarrow & \mathsf{if}\ 0\ \mathsf{then}\ (\mathsf{pred}\ (\mathsf{succ}\ 0))\ \mathsf{else}\ (\mathsf{succ}\ 0), \end{array}$$

There are no rules for evaluating this program further.

This program is not yet a value.

The type system of a type-safe language should reject such programs.

The Static Semantics: Typing

A type environment Γ is of the form $x_1 : T_1, \ldots, x_n : T_n$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{(T-Var)}$$

The Static Semantics: Typing

A type environment Γ is of the form $x_1 : T_1, \ldots, x_n : T_n$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{(T-Var)} \qquad\qquad \Gamma \vdash 0 : \mathsf{Int}\ \text{(T-Zero)}$$

The Static Semantics: Typing

A type environment Γ is of the form $x_1 : T_1, \ldots, x_n : T_n$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{(T-Var)} \qquad\qquad \Gamma \vdash \mathsf{0} : \mathsf{Int}\ \text{(T-Zero)}$$

$$\frac{\Gamma \vdash t\ : \mathsf{Int}}{\Gamma \vdash \mathsf{succ}\ t\ : \mathsf{Int}}\ \text{(T-Succ)}$$

The Static Semantics: Typing

A type environment Γ is of the form $x_1 : T_1, \ldots, x_n : T_n$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{(T-Var)} \qquad \Gamma \vdash 0 : \mathsf{Int} \text{ (T-Zero)}$$

$$\frac{\Gamma \vdash t : \mathsf{Int}}{\Gamma \vdash \mathsf{succ}\ t : \mathsf{Int}} \text{(T-Succ)} \qquad \frac{\Gamma \vdash t : \mathsf{Int}}{\Gamma \vdash \mathsf{pred}\ t : \mathsf{Int}} \text{(T-Pred)}$$

The Static Semantics: Typing

A type environment Γ is of the form $x_1 : T_1, \ldots, x_n : T_n$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{(T-Var)}$$

$$\Gamma \vdash 0 : \mathsf{Int} \text{ (T-Zero)}$$

$$\frac{\Gamma \vdash t \ : \mathsf{Int}}{\Gamma \vdash \mathsf{succ}\ t \ : \mathsf{Int}} \text{(T-Succ)}$$

$$\frac{\Gamma \vdash t \ : \mathsf{Int}}{\Gamma \vdash \mathsf{pred}\ t \ : \mathsf{Int}} \text{(T-Pred)}$$

$$\Gamma \vdash \mathsf{true} \ : \mathsf{Bool} \text{ (T-True)}$$

The Static Semantics: Typing

A type environment Γ is of the form $x_1 : T_1, \ldots, x_n : T_n$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-Var)}$$

$$\Gamma \vdash \mathsf{0} : \mathsf{Int} \text{ (T-Zero)}$$

$$\frac{\Gamma \vdash t \ : \mathsf{Int}}{\Gamma \vdash \mathsf{succ}\ t \ : \mathsf{Int}} \text{ (T-Succ)}$$

$$\frac{\Gamma \vdash t \ : \mathsf{Int}}{\Gamma \vdash \mathsf{pred}\ t \ : \mathsf{Int}} \text{ (T-Pred)}$$

$$\Gamma \vdash \mathsf{true} \ : \mathsf{Bool} \text{ (T-True)}$$

$$\Gamma \vdash \mathsf{false} \ : \mathsf{Bool} \text{ (T-False)}$$

The Static Semantics: Typing

A type environment Γ is of the form $\quad x_1 : T_1, \ldots, x_n : T_n$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-Var)} \qquad \Gamma \vdash \mathsf{0} : \mathsf{Int} \text{ (T-Zero)}$$

$$\frac{\Gamma \vdash t \ : \mathsf{Int}}{\Gamma \vdash \mathsf{succ}\ t \ : \mathsf{Int}} \text{ (T-Succ)} \qquad \frac{\Gamma \vdash t \ : \mathsf{Int}}{\Gamma \vdash \mathsf{pred}\ t \ : \mathsf{Int}} \text{ (T-Pred)}$$

$$\Gamma \vdash \mathsf{true} \ : \mathsf{Bool} \text{ (T-True)} \qquad \Gamma \vdash \mathsf{false} \ : \mathsf{Bool} \text{ (T-False)}$$

$$\frac{\Gamma \vdash t \ : \mathsf{Int}}{\Gamma \vdash \mathsf{iszero}\ t \ : \mathsf{Bool}} \text{ (T-IsZero)}$$

The Static Semantics: Typing

A type environment Γ is of the form $\quad x_1 : T_1, \ldots, x_n : T_n$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-Var)} \qquad \Gamma \vdash \mathsf{0} : \mathsf{Int} \text{ (T-Zero)}$$

$$\frac{\Gamma \vdash t : \mathsf{Int}}{\Gamma \vdash \mathsf{succ}\ t : \mathsf{Int}} \text{ (T-Succ)} \qquad \frac{\Gamma \vdash t : \mathsf{Int}}{\Gamma \vdash \mathsf{pred}\ t : \mathsf{Int}} \text{ (T-Pred)}$$

$$\Gamma \vdash \mathsf{true} : \mathsf{Bool} \text{ (T-True)} \qquad \Gamma \vdash \mathsf{false} : \mathsf{Bool} \text{ (T-False)}$$

$$\frac{\Gamma \vdash t : \mathsf{Int}}{\Gamma \vdash \mathsf{iszero}\ t : \mathsf{Bool}} \text{ (T-IsZero)} \qquad \frac{\Gamma \vdash t : \mathsf{Bool} \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 : T} \text{ (T-If)}$$

$$\frac{\Gamma, x : T \vdash t \;\; : T'}{\Gamma \vdash \mathsf{fun}\ x : T \cdot t \;\; : T \rightarrow T'} \text{ (T-Fun)}$$

$$\frac{\Gamma, x : T \vdash t \ : T'}{\Gamma \vdash \textsf{fun}\ x : T \cdot t \ : T \rightarrow T'}\ \text{(T-Fun)}$$

$$\frac{\Gamma \vdash t_1 \ : T \rightarrow T' \quad \Gamma \vdash t_2 \ : T}{\Gamma \vdash \textsf{apply}\ (t_1 \ , t_2 \ ) : T'}\ \text{(T-App)}$$

Example

$$\dfrac{\begin{array}{c}\vdots\\ \vdash \mathsf{fun}\ x : \mathsf{Bool}\cdot \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pred}\ 0)\ \mathsf{else}\ (\mathsf{succ}\ 0) : \mathsf{Bool}\to\mathsf{Int}\end{array} \qquad \dfrac{\dfrac{}{\vdash 0 : \mathsf{Int}}\,(\text{T-Zero})}{\vdash \mathsf{iszero}\ 0 : \mathsf{Bool}}\,(\text{T-IsZero})}{\vdash \mathsf{apply}\ (\mathsf{fun}\ x : \mathsf{Bool}\cdot \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pred}\ 0)\ \mathsf{else}\ (\mathsf{succ}\ 0), \mathsf{iszero}\ 0) : \mathsf{Int}}\,(\text{T-App})$$

Example

$$\dfrac{\dfrac{x : \mathsf{Bool} \vdash \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pred}\ 0)\ \mathsf{else}\ (\mathsf{succ}\ 0) : \mathsf{Int}}{}\!\!}{} $$

$$\dfrac{\dfrac{\dfrac{}{x : \mathsf{Bool} \vdash x : \mathsf{Bool}}\,(\text{T-Var}) \qquad \dfrac{x : \mathsf{Bool} \vdash 0 : \mathsf{Int}}{x : \mathsf{Bool} \vdash (\mathsf{pred}\ 0) : \mathsf{Int}}\,(\text{T-Pred}) \qquad \dfrac{x : \mathsf{Bool} \vdash 0 : \mathsf{Int}}{x : \mathsf{Bool} \vdash \mathsf{succ}\ 0 : \mathsf{Int}}\,(\text{T-Succ})}{x : \mathsf{Bool} \vdash \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pred}\ 0)\ \mathsf{else}\ (\mathsf{succ}\ 0) : \mathsf{Int}}\,(\text{T-If})}{\vdash \mathsf{fun}\ x\ : \mathsf{Bool} \cdot \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pred}\ 0)\ \mathsf{else}\ (\mathsf{succ}\ 0) : \mathsf{Bool} \rightarrow \mathsf{Int}}\,(\text{T-Fun})$$

$$\dfrac{\begin{array}{c}\vdots \\ \vdash \mathsf{fun}\ x\ : \mathsf{Bool} \cdot \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pred}\ 0)\ \mathsf{else}\ (\mathsf{succ}\ 0) : \mathsf{Bool} \rightarrow \mathsf{Int}\end{array} \qquad \dfrac{\dfrac{}{\vdash 0 : \mathsf{Int}}\,(\text{T-Zero})}{\vdash \mathsf{iszero}\ 0 : \mathsf{Bool}}\,(\text{T-IsZero})}{\vdash \mathsf{apply}\ (\mathsf{fun}\ x : \mathsf{Bool} \cdot \mathsf{if}\ x\ \mathsf{then}\ (\mathsf{pred}\ 0)\ \mathsf{else}\ (\mathsf{succ}\ 0), \mathsf{iszero}\ 0) : \mathsf{Int}}\,(\text{T-App})$$

The following program

$$\textsf{if true then } (\textsf{succ } 0) \textsf{ else } (\textsf{iszero } 0)$$

evaluates to $(\textsf{succ } 0)$ (doesn't get stuck).

However it is not well-typed according to our type system, i.e. we cannot show

$$\vdash \textsf{if true then } (\textsf{succ } 0) \textsf{ else } (\textsf{iszero } 0) : T$$

for any type T.

$\Longrightarrow$ we reject some safe programs.

The only required property for type safety is that all unsafe programs should be rejected.

The standard method for showing type safety.

(1) Progress

If $\vdash t : T$ and t is not a value then $t \longrightarrow t'$ for some term t' .

Well typed programs so not get stuck in some undefined state.

(2) Preservation

If $\vdash t : T$ and $t \longrightarrow t'$ then $\vdash t' : T$.

Evaluation preserves well-typedness (and type) of a program.

The proofs are usually easy (but long) once the right definitions have been found out.

Examples of type-safe languages: Java, SML.

Examples of type-unsafe languages: C, C++.

Progress: If $\vdash t : T$ and t is not a value then $t \longrightarrow t'$ for some term t'

Progress: If $\vdash t : T$ and t is not a value then $t \longrightarrow t'$ for some term t'

Proof: We do induction on the size of typing derivations.

– If t is true , false , $\mathsf{0}$ or $\mathsf{fun}\ x : T \cdot t'$ then there is nothing to prove because these are values.

$$\boxed{\text{Progress: If } \vdash t : T \text{ and } t \text{ is not a value then } t \longrightarrow t' \text{ for some term } t'}$$

Proof: We do induction on the size of typing derivations.

– If t is true , false , 0 or fun $x : T \cdot t'$ then there is nothing to prove because these are values.

– t cannot be a variable because the only rule for typing a variable is

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{(T-Var)}$$

which requires Γ to be non-empty.

some intersting cases:

– If t is of the form $\mathsf{succ}\ t'$, the typing derivation must be

$$\frac{\vdash t' : \mathsf{Int}}{\vdash \mathsf{succ}\ t' : \mathsf{Int}}\ \text{(T-Succ)}$$

If t' is a value then t is also a value. Othwerwise by induction hypothesis we have

$$\frac{t' \longrightarrow t''}{\mathsf{succ}\ t' \longrightarrow \mathsf{succ}\ t''}\ \text{(E-Succ)}$$

– If t is of the form $\mathsf{pred}\ t'$ then the typing derivation must be

$$\frac{\vdash t' : \mathsf{Int}}{\vdash \mathsf{pred}\ t' : \mathsf{Int}}\ \text{(T-Pred)}$$

(1) If t' is value 0 then by (E-PredZero) we know that $\mathsf{pred}\ t' \longrightarrow 0$.

(2) If t' is value $\mathsf{succ}\ nv$ then by (E-PredSucc) we know that $\mathsf{pred}\ t' \longrightarrow nv$.

(3) Otherwise t' is not a value. Hence by induction hypothesis we have

$$\frac{t' \longrightarrow t''}{\mathsf{pred}\ t' \longrightarrow \mathsf{pred}\ t''}\ \text{(E-Pred)}$$

– If t is of the form $\mathsf{iszero}\ t'$ then the typing derivation must be

$$\frac{\vdash t : \mathsf{Int}}{\vdash \mathsf{iszero}\ t : \mathsf{Bool}}\ \text{(T-IsZero)}$$

(1) If t' is value $\mathsf{0}$ then by (E-IsZeroZero) we know that $\mathsf{iszero}\ t' \longrightarrow \mathsf{true}$.

(2) If t' is value $\mathsf{succ}\ nv$ then by (E-IsZeroSucc) we know that $\mathsf{iszero}\ t' \longrightarrow \mathsf{false}$

(3) Otherwise t' is not a value and by induction hypothesis we have

$$\frac{t' \longrightarrow t''}{\mathsf{iszero}\ t' \longrightarrow \mathsf{iszero}\ t''}\ \text{(E-IsZero)}$$

– If t is of the form $\textsf{if } t_1 \textsf{ then } t_2 \textsf{ else } t_3$ then the typing derivation must be

$$\frac{\vdash t_1 : \textsf{Bool} \quad \vdash t_2 : T \quad \vdash t_3 : T}{\vdash \textsf{if } t_1 \textsf{ then } t_2 \textsf{ else } t_3 : T} \text{ (T-If)}$$

(1) If t_1 is value $\textsf{true}$ then by (E-IfTrue) we know that $t \longrightarrow t_2$.

(2) If t_1 is value $\textsf{false}$ then by (E-IfFalse) we know that $t \longrightarrow t_3$.

(3) Otherwise t_1 is not a value and by induction hypothesis we have

$$\frac{t_1 \longrightarrow t_1'}{\textsf{if } t_1 \textsf{ then } t_2 \textsf{ else } t_3 \longrightarrow \textsf{if } t_1' \textsf{ then } t_2 \textsf{ else } t_3} \text{ (E-If)}$$

– If t is of the form $\mathsf{apply}\ (t_1, t_2)$ then the typing derivation must be

$$\frac{\vdash t_1 : T \to T' \qquad \vdash t_2 : T}{\vdash \mathsf{apply}\ (t_1, t_2) : T'}\ \text{(T-App)}$$

(1) If t_1 is not a value then by induction hypothesis we have

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{apply}\ (t_1, t_2) \longrightarrow \mathsf{apply}\ (t_1', t_2)}\ \text{(E-App1)}$$

(2) If t_1 is value v_1 and t_2 is not a value then by induction hypothesis we have

$$\frac{t_2 \longrightarrow t_2'}{\mathsf{apply}\ (v_1, t_2) \longrightarrow \mathsf{apply}\ (v_1, t_2')}\ \text{(E-App2)}$$

(3) Suppose t_1 is a value and t_2 is also a value v_2 . Since $\vdash t_1 : T \to T'$ the value t_1 must be $\mathsf{fun}\ x : T \cdot t_1'$. Hence by (E-App) we have

$$\mathsf{apply}\ (\mathsf{fun}\ x : T \cdot t_1' , v_2) \longrightarrow t_1' [x \mapsto v_2]$$

:-)

$\boxed{\text{Preservation: If } \vdash t : T \text{ and } t \longrightarrow t' \text{ then } \vdash t' : T}$

Preservation: If $\vdash t : T$ and $t \longrightarrow t'$ then $\vdash t' : T$

Proof: induction on typing derivations.

Some interesting cases:

– t is of the form $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$. The typing derivation is of the form

$$\frac{\vdash t_1 : \mathsf{Bool} \qquad \vdash t_2 : T \qquad \vdash t_3 : T}{\vdash \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T}\ (\text{T-If})$$

(1) Suppose $t_1 \longrightarrow t_1'$ so that $t \longrightarrow t'$ wheret' is $\mathsf{if}\ t_1'\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$.

By induction hypothesis we know that $\Gamma \vdash t_1' : \mathsf{Bool}$ so that $\Gamma \vdash t' : T$.

(2) Suppose t_1 is true so that $t \longrightarrow t_2$ then we know that $\Gamma \vdash t_2 : T$.

– t is $\mathsf{apply}\ (\mathsf{fun}\ x : T' \cdot t_1\ , v_2)$ and the typing derivation is

$$\cfrac{\cfrac{x : T' \vdash t_1\ : T}{\vdash \mathsf{fun}\ x : T' \cdot t_1\ : T' \to T}\,(\text{T-Fun}) \qquad \vdash v_2 : T'}{\vdash \mathsf{apply}\ (\mathsf{fun}\ x : T' \cdot t_1\ , v_2) : T}\,(\text{T-App})$$

We have $t\ \longrightarrow t'$ where t' is $t_1\ [x \mapsto v_2]$.

To show that $\vdash t'\ : T$ we prove

Preservation of types under substitution

$$\boxed{\text{If } \Gamma, x : T' \vdash t_1\ : T \text{ and } \Gamma \vdash t_2\ : T' \text{ then } \Gamma \vdash t_1\ [x \mapsto t_2\ ] : T.}$$

Suppose now we extend the language by adding vectors.

$$\begin{array}{rcll} t & ::= & x \mid 0 & \\ & \mid & \ldots & \\ & \mid & [t\ , \ldots , t\ ] & \text{a vector of terms} \\ & \mid & \mathsf{get}\ t\ t & \text{accessing some ith element of a vector} \end{array}$$

Suppose now we extend the language by adding vectors.

$$
\begin{array}{rcll}
t & ::= & x \mid 0 & \\
 & \mid & \ldots & \\
 & \mid & [t\ ,\ldots,t\ ] & \text{a vector of terms} \\
 & \mid & \mathsf{get}\ t\ t & \text{accessing some ith element of a vector}
\end{array}
$$

Values $v ::= nv \mid \mathsf{true} \mid \mathsf{false} \mid [v,\ldots,v]$.

Suppose now we extend the language by adding vectors.

$$\begin{array}{rcll} t & ::= & x \mid 0 & \\ & \mid & \ldots & \\ & \mid & [t\ , \ldots, t\ ] & \text{a vector of terms} \\ & \mid & \mathsf{get}\ t\ t & \text{accessing some ith element of a vector} \end{array}$$

Values $v ::= nv \mid \mathsf{true} \mid \mathsf{false} \mid [v, \ldots, v]$.

Types $T ::= \mathsf{Int} \mid \mathsf{Bool} \mid T \to T \mid (\mathsf{vector}\ T)$

Suppose now we extend the language by adding vectors.

$$\begin{array}{lll} t ::= & x \mid 0 & \\ \mid & \ldots & \\ \mid & [t, \ldots, t] & \text{a vector of terms} \\ \mid & \mathsf{get}\ t\ t & \text{accessing some ith element of a vector} \end{array}$$

Values $v ::= nv \mid \mathsf{true} \mid \mathsf{false} \mid [v, \ldots, v]$.

Types $T ::= \mathsf{Int} \mid \mathsf{Bool} \mid T \to T \mid (\mathsf{vector}\ T)$

New evaluation rules

$$\frac{t_i \longrightarrow t_i'}{[v_0, \ldots, v_{i-1}, t_i, t_{i+1}, \ldots, t_n] \longrightarrow [v_0, \ldots, v_{i-1}, t_i', t_{i+1}, \ldots, t_n]} \text{(E-Vec)}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{get}\ t_1\ t_2 \longrightarrow \mathsf{get}\ t_1'\ t_2}\ (\text{E-Get1}) \qquad \frac{t_2 \longrightarrow t_2'}{\mathsf{get}\ v_1\ t_2 \longrightarrow \mathsf{get}\ v_1\ t_2'}\ (\text{E-Get2})$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{get}\ t_1\ t_2 \longrightarrow \mathsf{get}\ t_1'\ t_2}\ \text{(E-Get1)}$$

$$\frac{t_2 \longrightarrow t_2'}{\mathsf{get}\ v_1\ t_2 \longrightarrow \mathsf{get}\ v_1\ t_2'}\ \text{(E-Get2)}$$

$$\frac{i \leq n}{\mathsf{get}\ \mathsf{succ}\ ^{i}(0)\ [v_0, \ldots, v_n] \longrightarrow v_i}\ \text{(E-Get)}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathsf{get}\ t_1\ t_2 \longrightarrow \mathsf{get}\ t_1'\ t_2}\ \text{(E-Get1)} \qquad \frac{t_2 \longrightarrow t_2'}{\mathsf{get}\ v_1\ t_2 \longrightarrow \mathsf{get}\ v_1\ t_2'}\ \text{(E-Get2)}$$

$$\frac{i \leq n}{\mathsf{get}\ \mathsf{succ}\ ^{i}(0)\ [v_0, \ldots, v_n] \longrightarrow v_i}\ \text{(E-Get)}$$

New typing tules

$$\frac{\Gamma \vdash t_0\ : T \ldots \Gamma \vdash t_n\ : T}{\Gamma \vdash [t_0\ , \ldots, t_n\ ] : \mathsf{vector}\ T}\ \text{(T-Vec)} \qquad \frac{\Gamma \vdash t_1\ : \mathsf{Int} \quad \Gamma \vdash t_2\ : \mathsf{vector}\ T}{\Gamma \vdash \mathsf{get}\ t_1\ \ t_2\ : T}\ \text{(T-Get)}$$

Is the extended language type safe?

Is the extended language type safe?

No.

Preservation still holds, but progress fails.

Let term t be $\mathsf{get}\ (\mathsf{succ}\ (\mathsf{succ}\ (\mathsf{succ}\ 0)))[0,0]$. It is well-typed.

$$\dfrac{\begin{array}{c}\vdots\\ \vdash (\mathsf{succ}\ (\mathsf{succ}\ (\mathsf{succ}\ 0))) : \mathsf{Int}\end{array} \qquad \dfrac{\dfrac{}{\vdash 0 : \mathsf{Int}}\,(\text{T-Zero}) \qquad \dfrac{}{\vdash 0 : \mathsf{Int}}\,(\text{T-Zero})}{\vdash [0,0] : \mathsf{vector\ Int}}\,(\text{T-Vec})}{\vdash \mathsf{get}\ (\mathsf{succ}\ (\mathsf{succ}\ (\mathsf{succ}\ 0)))[0,0] : \mathsf{Int}}\,(\text{T-Get})$$

But there is no term t' such that $t \longrightarrow t'$.

Remedy 1: Modify the typing rules to reject such programs.

Problem: type inference involves problems like precise array bounds checking at compile time.

Remedy 1: Modify the typing rules to reject such programs.

Problem: type inference involves problems like precise array bounds checking at compile time.

Remedy 2: Modify the evaluation rules to take some necessary action in case of such ill-defined states.

Remedy 1: Modify the typing rules to reject such programs.

Problem: type inference involves problems like precise array bounds checking at compile time.

Remedy 2: Modify the evaluation rules to take some necessary action in case of such ill-defined states.

We introduce a new term for ill-defined states.

$$t \ ::= \ldots \mid error$$

Remedy 1: Modify the typing rules to reject such programs.

Problem: type inference involves problems like precise array bounds checking at compile time.

Remedy 2: Modify the evaluation rules to take some necessary action in case of such ill-defined states.

We introduce a new term for ill-defined states.

$$t \ ::= \ldots \mid \mathit{error}$$

and a rule for producing error message

$$\frac{i > n}{\textsf{get succ}\ ^{i}(0)\ [v_0, \ldots, v_n] \longrightarrow \mathit{error}}$$

and rules for propagating error messages

$\mathsf{apply}\ (error, t\ ) \longrightarrow error \qquad \mathsf{apply}\ (v, error) \longrightarrow error\ \ldots$

and rules for propagating error messages

$\mathsf{apply}\ (error, t\ ) \longrightarrow error \qquad \mathsf{apply}\ (v, error) \longrightarrow error\ \ldots$

Then we can show

Progress:

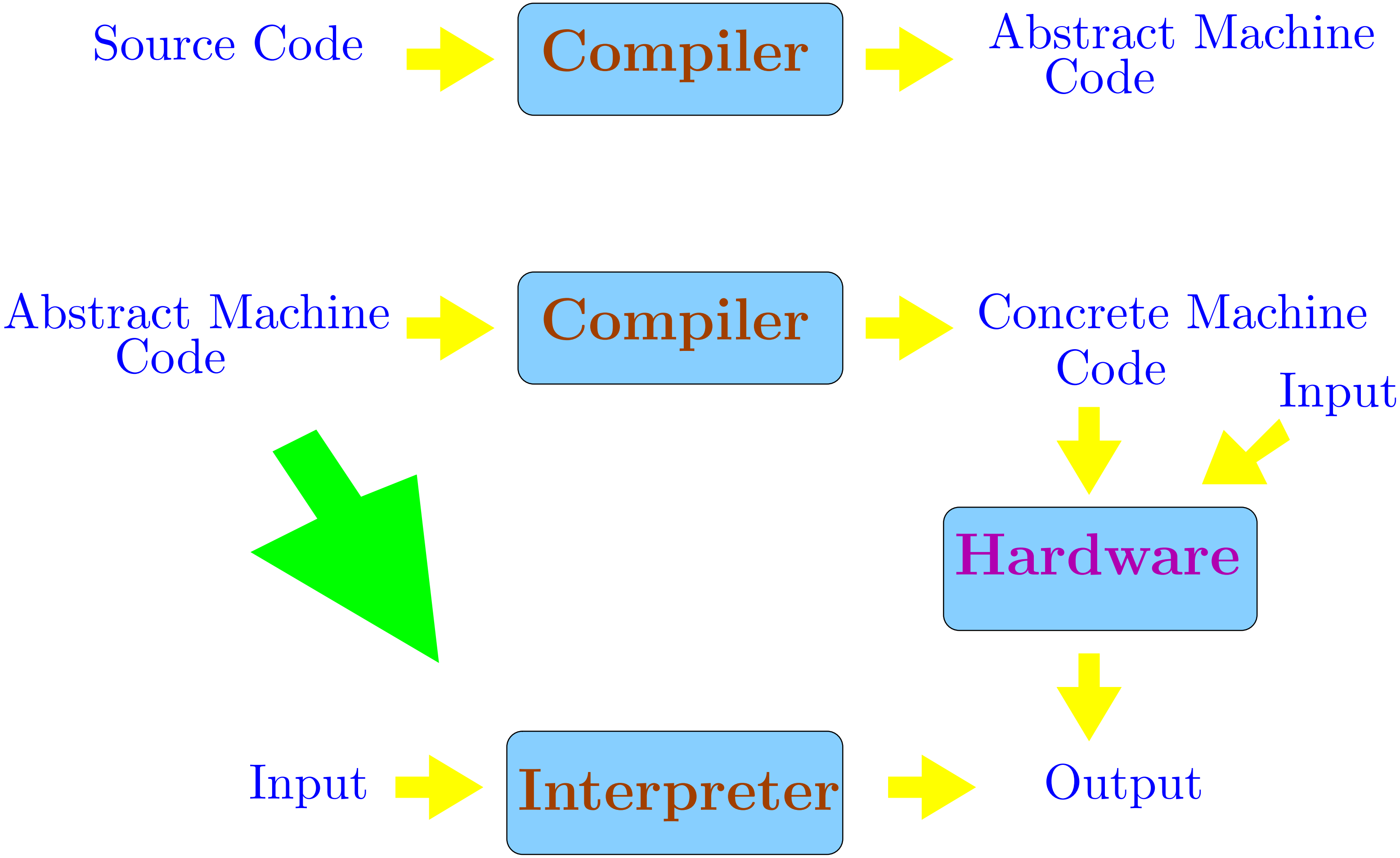
If $\vdash t\ : T$, t is not a value and $t\ \neq error$ then $t\ \longrightarrow t'$ for some t' .

Preservation:

If $\vdash t\ : T$ and $t\ \longrightarrow t'$ then either t' is $error$ or $\vdash t'\ : T$.

Java Security

The virtual machine principle:



Source Code

Compiler

Abstract Machine Code

Abstract Machine Code

Compiler

Concrete Machine Code

Input

Hardware

Input

Interpreter

Output

Java programs: definitions of classes.

```
public class hello {
  public static void main (String args[]) {
    System.out.println (”Hello!”); } }
```

Compilation produces class files containing Java bytecode.

```
javac hello.java
```

produces file hello.class containing bytecodes for the class hello.

A software implementing the Java Virtual Machine (JVM) executes the bytecodes to produce output.

```
java hello
```

Java programs: definitions of classes.

```
public class hello {
   public static void main (String args[]) {
      System.out.println (”Hello!”); } }
```

Compilation produces class files containing Java bytecode.

```
javac hello.java
```

produces file hello.class containing bytecodes for the class hello.

A software implementing the Java Virtual Machine (JVM) executes the bytecodes to produce output.

```
java hello
```

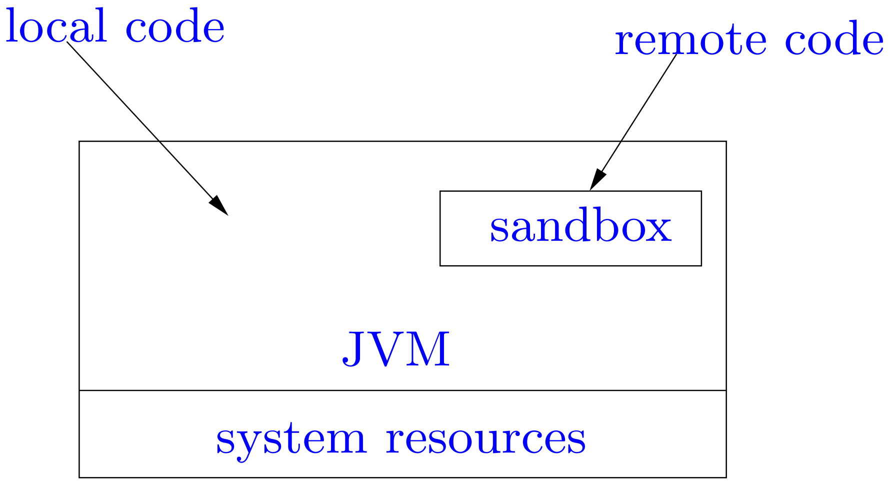
⟹ Portability

The sandbox principle: each application has access to a restricted set of system resources like local files, network, etc.

The sandbox principle: each application has access to a restricted set of system resources like local files, network, etc.

The original sandbox model

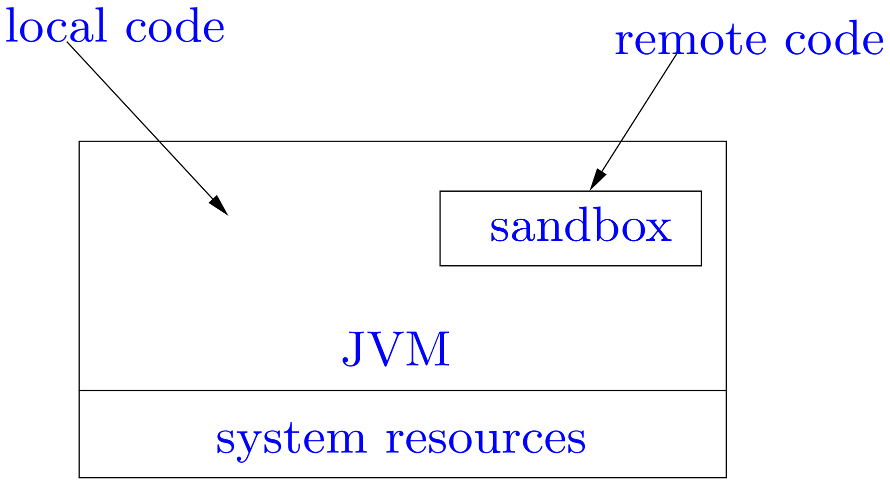
In JDK 1.0:



The sandbox principle: each application has access to a restricted set of system resources like local files, network, etc.

The original sandbox model

In JDK 1.0:



local code

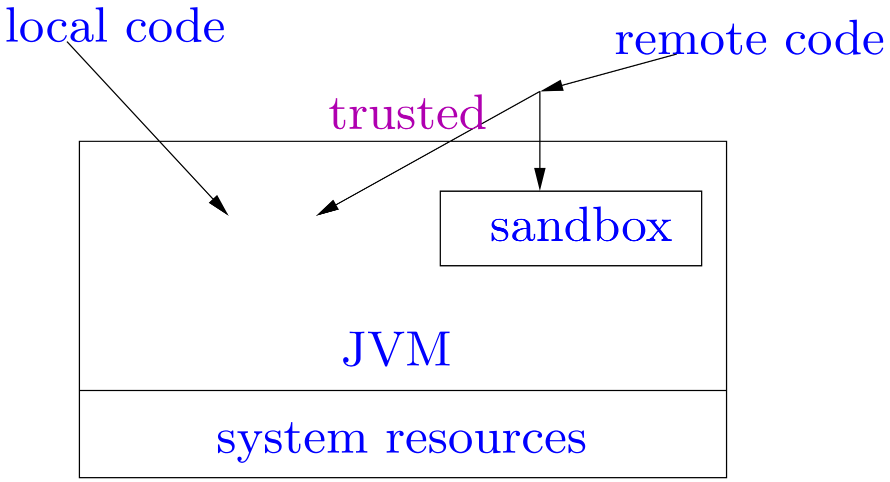
remote code

sandbox

JVM

system resources

In JDK 1.1:



local code

remote code

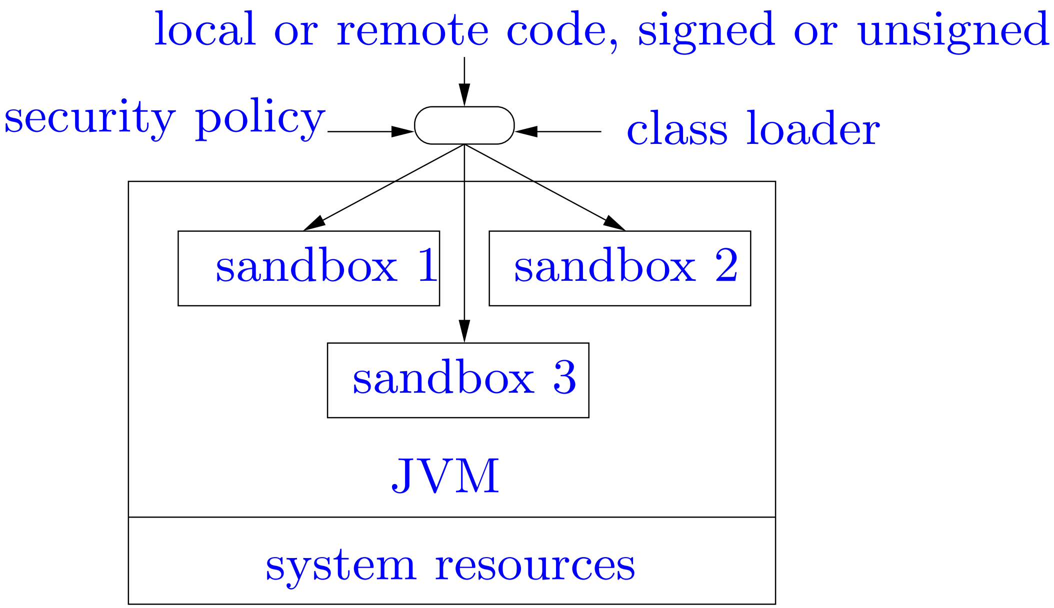
trusted

sandbox

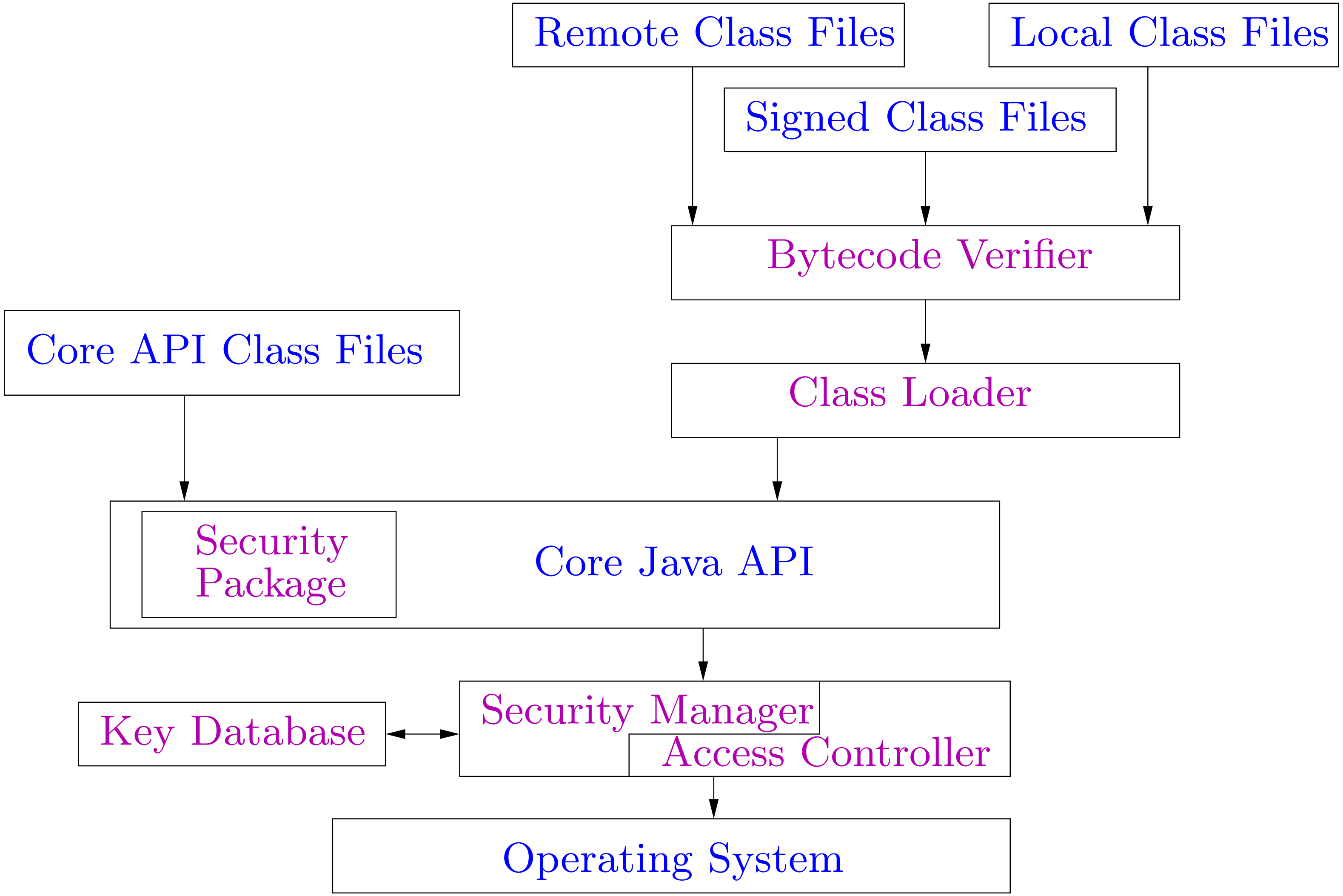
JVM

system resources

The current sandbox model in Java 2



Elements of the Java sandbox



Java language security constructs

Each entity has an access level

Specifier	Class	Package	Subclass	World
private	Yes	No	No	No
(Default)	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

Java language security constructs

Each entity has an access level

Specifier	Class	Package	Subclass	World
private	Yes	No	No	No
(Default)	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

Not sufficient for memory integrity . . .

- **No pointers:** prevents access to arbitrary memory locations.

- No pointers: prevents access to arbitrary memory locations.
- No use of variables before initialization.

- No pointers: prevents access to arbitrary memory locations.
- No use of variables before initialization.
- Array bounds checks.

- No pointers: prevents access to arbitrary memory locations.
- No use of variables before initialization.
- Array bounds checks.
- No arbitrary casts between different classes.

```
public class A {private int x;}
public class B {public int x;}
...
// a is of class A
B b = (B) a;
// The above is rejected by the compiler
Object o = b; B b' = o;
// The above is allowed by compiler but raises exception at runtime
```

Enforcement of the Java language rules.

- At compile time:

check typing rules, enforcement of access qualifiers, prevention of most illegal type casts.

- At load time:

verify bytecodes when a class is loaded (prevent malicious bytecodes)

- At runtime:

raise exceptions for illegal type casts, out of bound array accesses, . . .

Java Class Loading and Bytecode Verification

- Every object is a member of some class.

- The Class class: its members are the (definitions of) various classes that the JVM knows about.

- The classes can be dynamically loaded by the JVM by reading local or remote class files.

- Loading of classes is done by class loaders which are objects of the ClassLoader class.

- The class loader coordinates with the security manager and the access controller to provide the sandbox functions.

```
public class getClassTest {
  public static void main (String args[]) {
    String s = ”abc”;
    Class c1 = s.getClass();
    System.out.println (”string \”” + s + ”\” is of class ” + c1.getName());
    Class c2 = c1.getClass();
    System.out.println (”class ” + c1.getName() + ” is of class ” + c2.getName());
    Class c3 = c2.getClass();
    System.out.println (”class ” + c2.getName() + ” is of class ” + c3.getName());
  }
}
```

```
public class getClassTest {
  public static void main (String args[]) {
    String s = ”abc”;
    Class c1 = s.getClass();
    System.out.println (”string \”” + s + ”\” is of class ” + c1.getName());
    Class c2 = c1.getClass();
    System.out.println (”class ” + c1.getName() + ” is of class ” + c2.getName());
    Class c3 = c2.getClass();
    System.out.println (”class ” + c2.getName() + ” is of class ” + c3.getName());
  }
}
```

```
string ”abc” is of class java.lang.String
class java.lang.String is of class java.lang.Class
class java.lang.Class is of class java.lang.Class
```

An example involving dynamic class loading

```
import java.lang.reflect.*;

public class runhello {
  public static void main (String args[]) {
    Class c = null;
    Method m = null;

    // First we load the required class into the JVM
    try { c = Class.forName (”hello”);
    } catch (ClassNotFoundException e) {
      System.out.println (”The class was not found”);
    };
```

```
  // Get the main method of the class
  Class argtypes[] = new Class[] { String[].class };
  try { m = c.getMethod (”main”, argtypes);
  } catch (NoSuchMethodException e) {
    System.out.println (”The main method was not found”);
  };

  // Invoke the method
  Object arglist[] = new Object[1];
  try { m.invoke (null, arglist);
  } catch (Exception e) {
    System.out.println (”Error upon invocation” + e);
  };
} }
```

```
Hello!
```

The forName function finds, loads and links the class specified by the name.

forName(String name, boolean initialize, ClassLoader loader)

tries to find the class specified by the name, load it using the specified class loader and link it. The class is initialized if asked for.

forName (”hello”)

above is equivalent to

forName (”hello”, true, this.getClass().getClassLoader())

Security and the class loader

The security manager and access controller allow or prevent various operations depending upon the context of the request.

This information is provided by the class loader.

The class loader has information about

- origin: where the class was loaded from
- whether the class comes from the local filesystem or from the network
- whether the class comes with a digital signature

Each class loader defines a name space.

All classes loaded by particular class loader belong to its name space.

class loader cl1	class loader cl2	
java.lang.String	java.lang.String	. . .
abc	xyz	
	. . .	

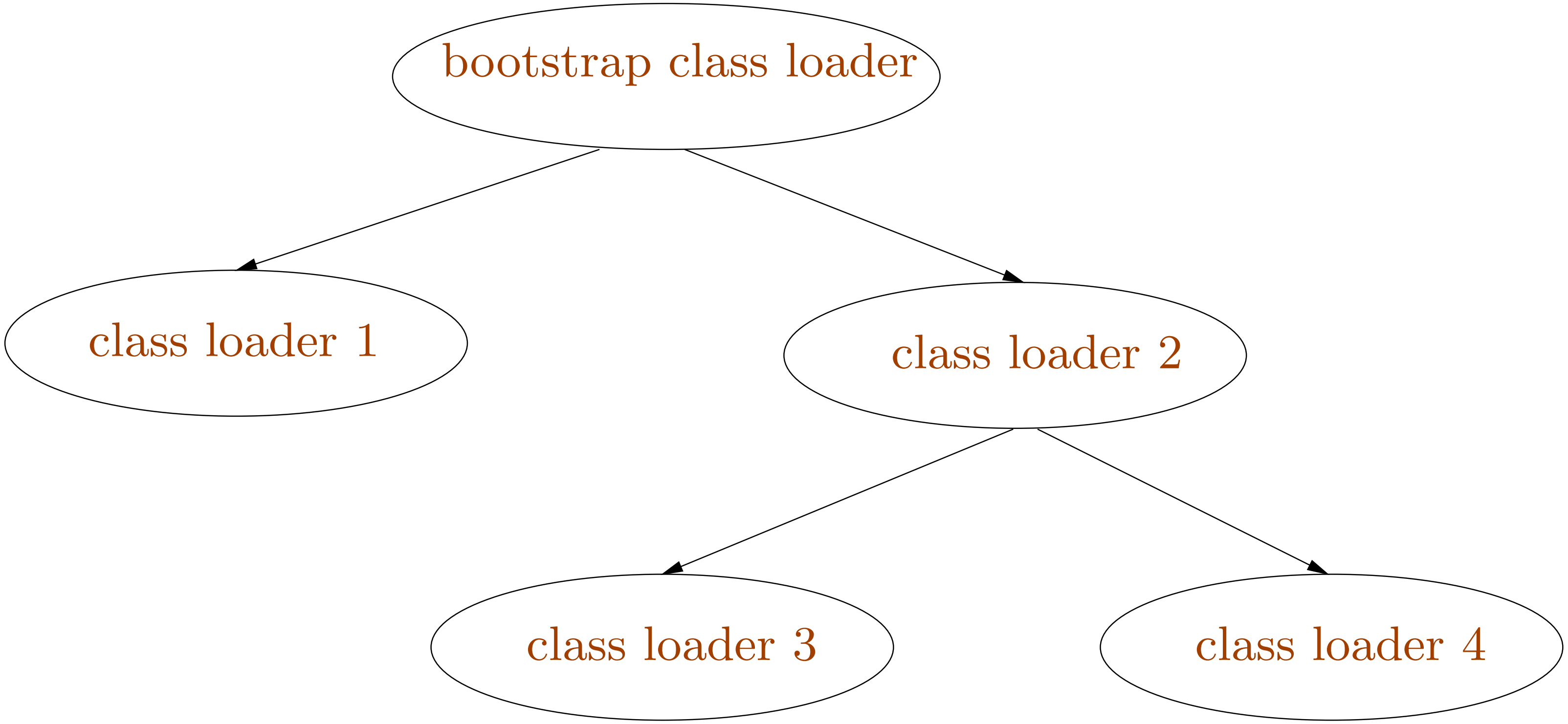
Classes from different sites are always loaded by different class loaders.

Hence the class java.lang.String provided by www.site1.com is different from the class java.lang.String provided by www.site2.com.

In particular they belong to different packages.

Hierarchy of class loaders

- The bootstrap class loader (primordial class loader, internal class loader) is responsible for loading a few initial classes when the JVM is launched.
- All new user defined class loaders have a parent class loader.



The class loading mechanism

1. return already existing class object, if found
2. ask the security manager for permission to access this class
3. attempt to load the class using the parent class loader
4. ask the security manager for permission to create this class
5. read the class file into an array of bytes
6. perform bytecode verification
7. create the class object
8. resolve the class

The class loading mechanism

1. return already existing class object, if found
2. ask the security manager for permission to access this class
3. attempt to load the class using the parent class loader
4. ask the security manager for permission to create this class
5. read the class file into an array of bytes
6. perform bytecode verification
7. create the class object
8. resolve the class

The mechanism can be overridden by class loaders in current versions of Java.

Using a class loader

Class loaders are members of (subclasses of) the abstract ClassLoader class.

Classes are loaded using the loadClass function of class loaders:

```
protected Class loadClass (String name, boolean resolve)
```

where name is the name of the class, and resolve tells us whether the class should be resolved or not.

Typically new classes of class loaders are defined by extending standard ones like SecureClassLoader or URLClassLoader.

Defining a new class of class loader

Either extend ClassLoader or one of its subclasses.

```
import java.net.∗;
// a trivial extension of URLClassLoader
public class myClassLoader extends URLClassLoader {
  myClassLoader (URL url) { super (new URL[] {url}); }

  protected Class loadClass (String name, boolean resolve) {
    Class c = null;
    try { c = super.loadClass(name, resolve);
    } catch (ClassNotFoundException e) { System.out.println (”Class not found”); }
    return c;
  }
}
```

Using a class loader

```
import java.lang.reflect.∗;
import java.net.∗;

public class runClass {
  public static void main (String args[]) {

    // Create a class loader
    URL url = null;
    try { url = new URL (”file:/home/userxyz/classes”);
    } catch (MalformedURLException e) { }
    myClassLoader cl = new myClassLoader(url);
```

```
    Class c = null; Method m = null;
    c = cl.loadClass (args[0]); // Load the class

    //Compute the argument vector and invoke the main method
    Class argtypes[] = new Class[] { String[].class };
    try { m = c.getMethod (”main”, argtypes); } catch (NoSuchMethodException e) {
      System.out.println (”The main method was not found”); };
    Object arglist[] = new Object[1];
    arglist[0] = new String[args.length - 1];
    for (int i=0; i < args.length - 1; i++) ((String[])arglist[0])[i] = args[i+1];
    try { m.invoke (null, arglist); } catch (Exception e) {
        System.out.println (”Error upon invocation” + e); };
  }
}
```

Java Bytecode Verification

Static analysis of the bytecodes to ensure security properties like

- operations follow typing rules
- no illegal casts
- no conversion from integers to pointers
- no calling of directly private methods of another class
- no jumping into the middle of a method
- no confusion between data and code

The JVM

- Stack based abstract machine: operations pop arguments and push results
- A set of registers, typically used for local variables and parameters: accessed by load and store instructions
- Stack and registers are preserved across method calls
- For each method, the number of stack slots and registers is specified in the bytecode
- unconditional, conditional and multiway (switch) intra-procedural branches
- Exception handlers table of entries (pc_1, pc_2, C, h): if exception of class C is raised between locations pc_1 and pc_2, then handler is at location h.
- Most JVM instructions are typed.

Example bytecode

The source code:

```
public class test {
  public static int factorial (int n) {
    int res;
    for (res = 1; n > 0; n−−) res = res ∗ n;
    return res;
  }
}
```

and the JVM bytecode (shown by running `javap` on the class file)…

```
...
public  static  int  factorial(int);  2 stack  slots ,  2  registers
   0:   iconst_1        // push integer constant 1
   1:   istore_1        // store it in register 1 (res)
   2:   iload_0         // push register 0 (n)
   3:   ifle  16        // if negative or zero, goto 16
   6:   iload_1         // push register 1 (res)
   7:   iload_0         // push register 0 (n)
   8:   imul            // multiply
   9:   istore_1        // store in register 1 (res)
   10:  iinc  0,  −1    // increment register 0 (n) by -1
   13:  goto 2          // goto beginning of loop
   16:  iload_1         // load register 1 (res)
   17:  ireturn         // return this value
...
```

Some properties to be verified

- Type correctness: the arguments of an instruction are always of the right type.
- No stack overflow or underflow
- Code containment: the PC points within the code for the method, at the beginning of an instruction
- Register initialization before use
- Object initialization before use

Minimize runtime checks $\Longrightarrow$ efficient execution

Verification idea: type level abstract interpretation

Use types as the abstract values.

The partial ordering $\sqsubseteq$ on types is the subtype relation.

Hence for example $C \sqsubseteq D \sqsubseteq \mathsf{Object}$ if class C extends class D.

We introduce special types Null and $\top$ to abstract null pointers and uninitialized values. Also $T \sqsubseteq \top$ for every T.

An abstract stack S is a sequence of types.

The sequence $S = \mathsf{Int} \cdot \mathsf{Int} \cdot \mathsf{Bool}$ abstracts a stack having a Boolean at the bottom of the stack and just two integers above it.

An abstract register assignment R maps registers to types.

$$R : \{0, \ldots, M_{reg} - 1\} \to \mathcal{T}$$

where M_{reg} is the maximum number of registers and $\mathcal{T}$ is the set of types.

An abstract state is either $\perp$ (unreachable state) or (S, R) where S is an abstract stack and R is an abstract register assignment.

Executing instructions modifies the abstract state.

$$(S, R) \xrightarrow{\mathsf{iconst}\ n} (\mathsf{Int} \cdot S, R)$$

if $|S| < M_{stack}$ where M_{stack} is the maximum size of the stack

Executing instructions modifies the abstract state.

$$(S, R) \xrightarrow{\textsf{iconst}\ n} (\textsf{Int} \cdot S, R)$$

if $|S| < M_{stack}$ where M_{stack} is the maximum size of the stack

$$(\textsf{Int} \cdot \textsf{Int} \cdot S, R) \xrightarrow{\textsf{iadd}} (\textsf{Int} \cdot S, R)$$

Executing instructions modifies the abstract state.

$$(S, R) \xrightarrow{\textsf{iconst}\ n} (\textsf{Int} \cdot S, R)$$

if $|S| < M_{stack}$ where M_{stack} is the maximum size of the stack

$$(\textsf{Int} \cdot \textsf{Int} \cdot S, R) \xrightarrow{\textsf{iadd}} (\textsf{Int} \cdot S, R)$$

$$(S, R) \xrightarrow{\textsf{iload}\ n} (\textsf{Int} \cdot S, R)$$

if $0 \leq n < M_{reg}$ and $R(n) = \textsf{Int}$ and $|S| < M_{stack}$

Executing instructions modifies the abstract state.

$$(S, R) \xrightarrow{\mathsf{iconst}\ n} (\mathsf{Int} \cdot S, R)$$

if $|S| < M_{stack}$ where M_{stack} is the maximum size of the stack

$$(\mathsf{Int} \cdot \mathsf{Int} \cdot S, R) \xrightarrow{\mathsf{iadd}} (\mathsf{Int} \cdot S, R)$$

$$(S, R) \xrightarrow{\mathsf{iload}\ n} (\mathsf{Int} \cdot S, R)$$

if $0 \leq n < M_{reg}$ and $R(n) = \mathsf{Int}$ and $|S| < M_{stack}$

$$(\mathsf{Int} \cdot S, R) \xrightarrow{\mathsf{istore}\ n} (S, R\{n \mapsto \mathsf{Int}\})$$

if $0 \leq n < M_{reg}$

$$(\mathsf{Int} \cdot S, R) \xrightarrow{\mathsf{ifle}\ n} (S, R)$$

if n is a valid instruction location

$$(S, R) \xrightarrow{\mathsf{goto}\ n} (S, R)$$

if n is a valid instruction location

$$(S, R) \xrightarrow{\mathsf{aconst\_null}} (\mathsf{Null} \cdot S, R)$$

$$\text{if } |S| < M_{stack}$$

$$(S, R) \xrightarrow{\mathsf{aconst\_null}} (\mathsf{Null} \cdot S, R)$$

$$\text{if } |S| < M_{stack}$$

$$(S, R) \xrightarrow{\mathsf{aload}\ \ n} (R(n) \cdot S, R)$$

$$\text{if } 0 \leq n < M_{reg} \text{ and } R(n) \sqsubseteq \mathsf{Object} \text{ and } |S| < M_{stack}$$

$$(S, R) \xrightarrow{\mathsf{aconst\_null}} (\mathsf{Null} \cdot S, R)$$

$$\text{if } |S| < M_{stack}$$

$$(S, R) \xrightarrow{\mathsf{aload}\ n} (R(n) \cdot S, R)$$

$$\text{if } 0 \leq n < M_{reg} \text{ and } R(n) \sqsubseteq \mathsf{Object} \text{ and } |S| < M_{stack}$$

$$(\tau \cdot S, R) \xrightarrow{\mathsf{astore}\ n} (S, R\{n \mapsto \tau\})$$

$$\text{if } 0 \leq n < M_{reg} \text{ and } \tau \sqsubseteq \mathsf{Object}$$

Accessing fields and methods

$$(\tau' \cdot S, R) \xrightarrow{\textsf{getfield}\ C.f.\tau} (\tau \cdot S, R)$$

$$\text{if } \tau' \sqsubseteq C$$

Accessing fields and methods

$$(\tau' \cdot S, R) \xrightarrow{\textsf{getfield}\ C.f.\tau} (\tau \cdot S, R)$$

$$\text{if } \tau' \sqsubseteq C$$

$$(\tau_1 \cdot \tau_2 \cdot S, R) \xrightarrow{\textsf{putfield}\ C.f.\tau} (S, R)$$

$$\text{if } \tau_1 \sqsubseteq \tau \text{ and } \tau_2 \sqsubseteq C$$

Accessing fields and methods

$$(\tau' \cdot S, R) \xrightarrow{\mathsf{getfield}\ C.f.\tau} (\tau \cdot S, R)$$

$$\text{if } \tau' \sqsubseteq C$$

$$(\tau_1 \cdot \tau_2 \cdot S, R) \xrightarrow{\mathsf{putfield}\ C.f.\tau} (S, R)$$

$$\text{if } \tau_1 \sqsubseteq \tau \text{ and } \tau_2 \sqsubseteq C$$

$$(\tau'_n \cdot \ldots \cdot \tau'_1 \cdot S, R) \xrightarrow{\mathsf{invokestatic}\ C.m.\sigma} (\tau \cdot S, R)$$

$$\text{if } \sigma = \tau(\tau_1, \ldots, \tau_n), \tau'_i \sqsubseteq \tau_i \text{ for } 1 \leq i \leq n \text{ and } |\tau \cdot S| \leq M_{stack}$$

Accessing fields and methods

$$(\tau' \cdot S, R) \xrightarrow{\mathsf{getfield}\ C.f.\tau} (\tau \cdot S, R)$$

$$\text{if } \tau' \sqsubseteq C$$

$$(\tau_1 \cdot \tau_2 \cdot S, R) \xrightarrow{\mathsf{putfield}\ C.f.\tau} (S, R)$$

$$\text{if } \tau_1 \sqsubseteq \tau \text{ and } \tau_2 \sqsubseteq C$$

$$(\tau'_n \cdot \ldots \cdot \tau'_1 \cdot S, R) \xrightarrow{\mathsf{invokestatic}\ C.m.\sigma} (\tau \cdot S, R)$$

$$\text{if } \sigma = \tau(\tau_1, \ldots, \tau_n), \tau'_i \sqsubseteq \tau_i \text{ for } 1 \leq i \leq n \text{ and } |\tau \cdot S| \leq M_{stack}$$

$$(\tau'_n \cdot \ldots \cdot \tau'_1 \cdot \tau' \cdot S, R) \xrightarrow{\mathsf{invokevirtual}\ C.m.\sigma} (\tau \cdot S, R)$$

$$\text{if } \sigma = \tau(\tau_1, \ldots, \tau_n), \tau' \sqsubseteq C, \tau'_i \sqsubseteq \tau_i \text{ for } 1 \leq i \leq n \text{ and } |\tau \cdot S| \leq M_{stack}$$

Another example

```
public class testclass {
  public testclass () { }
  public Class testfunction (String s) {
    Class c = s.getClass();
    return c;
  }
}
```

```
public java.lang.Class testfunction(java.lang.String); 1 stack slots, 3 registers
   0: aload_1
   1: invokevirtual #2; //Method java/lang/Object.getClass:()Ljava/lang/Class;
   4: astore_2
   5: aload_2
   6: areturn
```

Our analysis on this example

```
public java.lang.Class testfunction(java.lang.String); 1 stack slots, 3 registers
                                // stack, R(0), R(1), R(2)
                                // ϵ, (testclass, String, ⊤)
   0:   aload_1                 // String, (testclass, String, ⊤)
   1:   invokevirtual #2;       // Class, (testclass, String, ⊤)
   4:   astore_2                // ϵ, (testclass, String, Class)
   5:   aload_2                 // Class, (testclass, String, Class)
   6:   areturn
```

In case of several paths to a node, we need to compute least upper bounds $\sqcup$.

Comparison of abstract stacks:

$$T_1 \cdot \ldots \cdot T_n \sqsubseteq U_1 \cdot \ldots \cdot U_n \quad \text{iff} \quad T_i \sqsubseteq U_i \text{ for } 1 \leq i \leq n.$$

$$T_1 \cdot \ldots \cdot T_n \sqcup U_1 \cdot \ldots \cdot U_n = T_1 \sqcup U_1 \cdot \ldots \cdot T_n \sqcup U_n$$

Comparison of abstract register assignments:

$$R_1 \sqsubseteq R_2 \quad \text{iff} \quad R_1(i) \sqsubseteq R_2(i) \text{ for } 0 \leq i < M_{reg}.$$

$$(R_1 \sqcup R_2)(n) = R_1(n) \sqcup R_2(n)$$

Comparison of abstract states

$$(S_1, R_1) \sqsubseteq (S_2, R_2) \quad \text{iff} \quad S_1 \sqsubseteq S_2 \text{ and } R_1 \sqsubseteq R_2$$

$$(S_1, R_1) \sqcup (S_2, R_2) = (S_1 \sqcup S_2, R_1 \sqcup R_2)$$

Also $\perp \sqsubseteq (R, S)$ and $\perp \sqsubseteq (R, S) = (R, S)$.

Initial abstract state: (S_{start}, R_{start}) where $S_{start} = \epsilon$ is the empty stack and $R_{start}(0), \ldots, R_{start}(n-1)$ are the n arguments, and $R_{start}(i) = \top$ for $i \geq n$

If $\pi : pc_1 \to pc_2$ is a path (possibly with loops) from pc_1 to pc_2 with corresponding instruction sequence $I_1, \ldots, I_k$ and

$$(R_{i-1}, S_{i-1}) \xrightarrow{I_i} (S_i, R_i)$$

for $1 \leq i \leq n$ then we write $\pi : (S_0, R_0) \to (S_k, R_k)$.

For every valid location pc we define

Merge Over All Paths (MOP):

$$\mathcal{S}[pc] = \bigsqcup\{(S, R) \mid \pi : (S_{start}, R_{start}) \to (S, R)\}$$

Example

Suppose classes D and E are defined by extending class C, so that $D \sqcup E = C$.

```
                        // Int, (D, E)
10:  ifle  17           // ϵ, (D, E)
13:  aload_0            // D, (D, E)
14:  goto 18            // ϵ, (D, E)
17:  aload_1            // C, (D, E)
18:  areturn
```

(According to our notation, $C, (D, E)$ is the abstract state before the execution of the instruction at location 18.)

Another example

```
                      // ε, (Int, String)
9:   iload_0          // Int, (Int, String)
10:  ifle   17        // ε, (Int, String)
13:  iload_0          // Int, (Int, String)
14:  goto 18          // ε, (Int, String)
17:  aload_1          // ⊤, (Int, String)
18:  areturn
```

The bytecode verification fails because the return value is of unknown type.

```
public static int factorial(int); 2 stack slots, 2 registers
                              // ϵ, (Int, ⊤)
   0:  iconst_1               // Int, (Int, ⊤)
   1:  istore_1               // ϵ, (Int, Int)
   2:  iload_0                // Int, (Int, Int)
   3:  ifle 16                // ϵ, (Int, Int)
   6:  iload_1                // Int, (Int, Int)
   7:  iload_0                // Int · Int, (Int, Int)
   8:  imul                   // Int, (Int, Int)
   9:  istore_1               // ϵ, (Int, Int)
   10: iinc 0, −1             // ϵ, (Int, Int)
   13: goto 2                 // ϵ, (Int, Int)
   16: iload_1                // Int, (Int, Int)
   17: ireturn
```

Other issues to be tackled in the full Java bytecode language:

- initialization of objects
- exception handling

Typed Assembly Language (TAL)

Morrisett et al.

- A generic approach to safe compiled code.
- Based on the concept of type safety.
- Use type preserving compilation to transform type safe source code to type safe compiled code.
- Can be combined with the idea of proof carrying code.

A first language: TAL-0

Deals with control flow safety: no jumps to arbitrary machine addresses.

A first language: TAL-0

Deals with control flow safety: no jumps to arbitrary machine addresses.

Syntax of programs: We assume a fixed finite set of registers:

r ::=		registers
	r1 | . . . | rk	
ν ::=		operands
	n	integer
	| l	label
	| r	register

A first language: TAL-0

Deals with control flow safety: no jumps to arbitrary machine addresses.

Syntax of programs: We assume a fixed finite set of registers:

$r ::=$		registers
	$\mathsf{r1} \mid \ldots \mid \mathsf{rk}$	
$\nu ::=$		operands
	n	integer
	$\mid l$	label
	$\mid r$	register

$\iota ::=$		instructions
	$r_d := \nu$	
	$\mid r_d := r_s + \nu$	
	$\mid \mathsf{if}\ r\ \mathsf{jump}\ \nu$	

A first language: TAL-0

Deals with control flow safety: no jumps to arbitrary machine addresses.

Syntax of programs: We assume a fixed finite set of registers:

$r ::=$		registers
	$\mathsf{r1} \mid \ldots \mid \mathsf{rk}$	
$\nu ::=$		operands
	n	integer
	$\mid l$	label
	$\mid r$	register

$\iota ::=$		instructions
	$r_d := \nu$	
	$\mid r_d := r_s + \nu$	
	$\mid \mathsf{if}\ r\ \mathsf{jump}\ \nu$	
$I ::=$		instruction sequences
	$\mathsf{jump}\ \nu$	
	$\mid \iota; I$	

A first language: TAL-0

Deals with control flow safety: no jumps to arbitrary machine addresses.

Syntax of programs: We assume a fixed finite set of registers:

$r ::=$ registers

$\mathsf{r1} \mid \ldots \mid \mathsf{rk}$

$\nu ::=$ operands

n integer

$\mid l$ label

$\mid r$ register

$\iota ::=$ instructions

$r_d := \nu$

$\mid r_d := r_s + \nu$

$\mid \mathsf{if}\ r\ \mathsf{jump}\ \nu$

$I ::=$ instruction sequences

$\mathsf{jump}\ \nu$

$\mid \iota; I$

Operands other than registers are called values (i.e. registers and labels).

- Instruction sequences have an unconditional jump at the end, and other instructions before.
- As yet, no infinite memory (except for code).

- Instruction sequences have an unconditional jump at the end, and other instructions before.
- As yet, no infinite memory (except for code).

An example for computing product: r4 contains the return address

```
prod :   r3 := 0;
          jump loop
loop :   if r1 jump done;
         r3 := r2 + r3;
         r1 := r1 + −1;
          jump loop
done :    jump r4
```

The example has three instruction sequences, and a label corresponding to each of them.

Evaluation: the TAL-0 abstract machine

- the abstract machine contains the code and data.
- an evaluation step changes the state (code and data) of the abstract machine.

$R ::=$ register files

$\{\mathsf{r1} \mapsto \nu_1, \ldots, \mathsf{rk} \mapsto \nu_k\}$ (each ν_i is a value)

$h ::=$ heap values

I instruction sequences

$H ::=$ heaps

$\{l_1 \mapsto h_1, \ldots l_m \mapsto h_m\}$

$M ::=$ abstract machine states

(H, R, I) (I is the current instruction sequence being executed)

- A register file R maps each register r to some value (integer or label) $R(r)$.

- A register file R maps each register r to some value (integer or label) $R(r)$.
- A heap H is a partial map: H maps some labels l to heap values $H(l)$.

- A register file R maps each register r to some value (integer or label) $R(r)$.
- A heap H is a partial map: H maps some labels l to heap values $H(l)$.

The previous example has three instruction sequences

$I_1 = \mathsf{r3} := \mathsf{0};\ \mathsf{jump\ loop}$

$I_2 = \mathsf{if\ r1\ jump\ done}; \mathsf{r3} := \mathsf{r2} + \mathsf{r3}; \mathsf{r1} := \mathsf{r1} + -\mathsf{1};\ \mathsf{jump\ loop}$

$I_3 = \ \mathsf{jump\ r4}$

We have the heap $H_0 = \{\mathsf{prod} \mapsto I_1, \mathsf{loop} \mapsto I_2, \mathsf{done} \mapsto I_3\}$.

- A register file R maps each register r to some value (integer or label) $R(r)$.
- A heap H is a partial map: H maps some labels l to heap values $H(l)$.

The previous example has three instruction sequences

$I_1 = \mathsf{r3} := \mathsf{0};\ \mathsf{jump\ loop}$

$I_2 = \mathsf{if\ r1\ jump\ done}; \mathsf{r3} := \mathsf{r2} + \mathsf{r3}; \mathsf{r1} := \mathsf{r1} + -\mathsf{1};\ \mathsf{jump\ loop}$

$I_3 =\ \mathsf{jump\ r4}$

We have the heap $H_0 = \{\mathsf{prod} \mapsto I_1, \mathsf{loop} \mapsto I_2, \mathsf{done} \mapsto I_3\}$.

The starting state of the machine is supposed to be of the form

$$M_0 = (H_0, R_0, I_1)$$

where $R_0(\mathsf{r1}) = \mathsf{n}$ and $R_0(\mathsf{r2}) = \mathsf{m}$ are integers and $R_0(\mathsf{r4})$ is a label.

A possible execution sequence: …

$$
\begin{array}{llllll}
H_0, & \{r1 \mapsto \mathsf{2}, & r2 \mapsto \mathsf{2}, & r3 \mapsto \mathsf{0}, & r4 \mapsto \mathsf{l}\}, & \mathsf{r3} := \mathsf{0};\ \mathsf{jump\ loop} \\
H_0, & \{r1 \mapsto \mathsf{2}, & r2 \mapsto \mathsf{2}, & r3 \mapsto \mathsf{0} & r4 \mapsto \mathsf{l}\}, & \mathsf{jump\ loop} \\
H_0, & \{r1 \mapsto \mathsf{2}, & r2 \mapsto \mathsf{2}, & r3 \mapsto \mathsf{0} & r4 \mapsto \mathsf{l}\}, & I_2 \\
H_0, & \{r1 \mapsto \mathsf{2}, & r2 \mapsto \mathsf{2}, & r3 \mapsto \mathsf{0} & r4 \mapsto \mathsf{l}\}, & r3 := r2 + r3; \mathsf{r1} := \mathsf{r1} + \mathsf{-1};\ \mathsf{jump\ loop} \\
H_0, & \{r1 \mapsto \mathsf{2}, & r2 \mapsto \mathsf{2}, & r3 \mapsto \mathsf{2} & r4 \mapsto \mathsf{l}\}, & \mathsf{r1} := \mathsf{r1} + \mathsf{-1};\ \mathsf{jump\ loop} \\
H_0, & \{r1 \mapsto \mathsf{1}, & r2 \mapsto \mathsf{2}, & r3 \mapsto \mathsf{2} & r4 \mapsto \mathsf{l}\}, & \mathsf{jump\ loop} \\
H_0, & \{r1 \mapsto \mathsf{1}, & r2 \mapsto \mathsf{2}, & r3 \mapsto \mathsf{2} & r4 \mapsto \mathsf{l}\}, & I_2 \\
H_0, & \{r1 \mapsto \mathsf{1}, & r2 \mapsto \mathsf{2}, & r3 \mapsto \mathsf{2} & r4 \mapsto \mathsf{l}\}, & r3 := r2 + r3; \mathsf{r1} := \mathsf{r1} + \mathsf{-1};\ \mathsf{jump\ loop} \\
H_0, & \{r1 \mapsto \mathsf{1}, & r2 \mapsto \mathsf{2}, & r3 \mapsto \mathsf{4} & r4 \mapsto \mathsf{l}\}, & \mathsf{r1} := \mathsf{r1} + \mathsf{-1};\ \mathsf{jump\ loop} \\
H_0, & \{r1 \mapsto \mathsf{0}, & r2 \mapsto \mathsf{2}, & r3 \mapsto \mathsf{4} & r4 \mapsto \mathsf{l}\}, & \mathsf{jump\ loop} \\
H_0, & \{r1 \mapsto \mathsf{0}, & r2 \mapsto \mathsf{2}, & r3 \mapsto \mathsf{4} & r4 \mapsto \mathsf{l}\}, & I_2 \\
H_0, & \{r1 \mapsto \mathsf{0}, & r2 \mapsto \mathsf{2}, & r3 \mapsto \mathsf{4} & r4 \mapsto \mathsf{l}\}, & \mathsf{jump}\ r4
\end{array}
$$

As usual, we formalize this using evaluation rules.

As usual, we formalize this using evaluation rules.

$$\frac{H(\hat{R}(\nu)) = I}{(H, R,\ \mathsf{jump}\ \nu) \longrightarrow (H, R, I)} \text{E-Jump)}$$

where the lookup function $\hat{R}$ returns the value corresponding to an operand:

$$\hat{R}(r) = R(r)$$

$$\hat{R}(n) = n$$

$$\hat{R}(l) = l$$

The JUMP instruction loads a new instruction sequence which should then be executed.

(The machine is stuck if $\hat{R}(\nu)$ is not a label.)

Otherwise, we consume one instruction from the current instruction sequence.

The MOV and ADD instructions modify the register file.

$$(H, R, r_d := \nu; I) \longrightarrow (H, R \oplus \{r_d \mapsto \hat{R}(\nu)\}, I) \qquad \text{(E-Mov)}$$

Otherwise, we consume one instruction from the current instruction sequence.

The MOV and ADD instructions modify the register file.

$$(H, R, r_d := \nu; I) \longrightarrow (H, R \oplus \{r_d \mapsto \hat{R}(\nu)\}, I) \qquad \text{(E-Mov)}$$

$$\frac{R(r_s) = n_1 \qquad \hat{R}(\nu) = n_2}{(H, R, r_d := r_s + \nu; I) \longrightarrow (H, R \oplus \{r_d \mapsto n_1 + n_2\}, I)} \text{(E-Add)}$$

(The machine is stuck in the second case if $R(r_s)$ or $\hat{R}(\nu)$ is not an integer.)

The conditional jump instruction either loads a new instruction sequence or just consumes one instruction.

$$\frac{R(r) = \mathsf{0} \qquad H(\hat{R}(\nu)) = I'}{(H, R, \mathsf{if}\ r\ \mathsf{jump}\ \nu; I) \longrightarrow (H, R, I')}\ \text{(E-IfEq)}$$

The conditional jump instruction either loads a new instruction sequence or just consumes one instruction.

$$\frac{R(r) = 0 \qquad H(\hat{R}(\nu)) = I'}{(H, R, \mathsf{if}\ r\ \mathsf{jump}\ \nu; I) \longrightarrow (H, R, I')}\ \text{(E-IfEq)}$$

$$\frac{R(r) = n \qquad n \neq 0}{(H, R, \mathsf{if}\ r\ \mathsf{jump}\ \nu; I) \longrightarrow (H, R, I)}\ \text{(E-IfNeq)}$$

(The machine is stuck if $R(r)$ is not an integer or, in the first case, if $\hat{R}(\nu)$ is not a label.)

Consider the following simple code:

```
l :  r1 := 5;
     jump r1
```

Consider the following simple code:

```
l :   r1 := 5;
      jump r1
```

Define instruction sequence $I = \mathsf{r1} := \mathsf{5};\ \mathsf{jump}\ \mathsf{r1}$ and heap $H = \{\mathsf{l} \mapsto I\}$.

Corresponding to the above code, starting with register file $R = \{\mathsf{r1} \mapsto \mathsf{0}\}$ we have the evaluation step

$$(H, \{\mathsf{r1} \mapsto \mathsf{0}\}, I) \longrightarrow (H, \{\mathsf{r1} \mapsto \mathsf{5}\},\ \mathsf{jump}\ \mathsf{r1})$$

Consider the following simple code:

```
l :  r1 := 5;
     jump r1
```

Define instruction sequence $I = \mathsf{r1} := \mathsf{5};\ \mathsf{jump}\ \mathsf{r1}$ and heap $H = \{\mathsf{l} \mapsto I\}$.

Corresponding to the above code, starting with register file $R = \{\mathsf{r1} \mapsto \mathsf{0}\}$ we have the evaluation step

$$(H, \{\mathsf{r1} \mapsto \mathsf{0}\}, I) \longrightarrow (H, \{\mathsf{r1} \mapsto \mathsf{5}\},\ \mathsf{jump}\ \mathsf{r1})$$

The machine is now stuck: no further evaluation step is possible because r1 stores an integer instead of a label.

Consider the following simple code:

$$
\begin{aligned}
\mathsf{l}: \quad & \mathsf{r1} := \mathsf{5}; \\
& \mathsf{jump\ r1}
\end{aligned}
$$

Define instruction sequence $I = \mathsf{r1} := \mathsf{5};\ \mathsf{jump\ r1}$ and heap $H = \{\mathsf{l} \mapsto I\}$.

Corresponding to the above code, starting with register file $R = \{\mathsf{r1} \mapsto \mathsf{0}\}$ we have the evaluation step

$$(H, \{\mathsf{r1} \mapsto \mathsf{0}\}, I) \longrightarrow (H, \{\mathsf{r1} \mapsto \mathsf{5}\},\ \mathsf{jump\ r1})$$

The machine is now stuck: no further evaluation step is possible because $\mathsf{r1}$ stores an integer instead of a label.

Hence to filter out such bad programs, we need to introduce typing rules.

Initial idea for a TAL-0 typing system: introduce two different types Int and Code for integers and labels.

In the previous example, we will start with the register file type $\Gamma = \{\mathsf{r1} : \mathsf{Int}\}$.

After the instruction $\mathsf{r1} = \mathsf{5}$ the register file type remains the same.

Then the second instruction $\mathsf{jump}\ \mathsf{r1}$ fails to type check because $\Gamma(\mathsf{r1})$ is Int instead of Code.

Hence the code is rejected, as desired.

Initial idea for a TAL-0 typing system: introduce two different types Int and Code for integers and labels.

In the previous example, we will start with the register file type $\Gamma = \{\mathsf{r1} : \mathsf{Int}\}$.

After the instruction $\mathsf{r1} = \mathsf{5}$ the register file type remains the same.

Then the second instruction $\mathsf{jump}\ \mathsf{r1}$ fails to type check because $\Gamma(\mathsf{r1})$ is Int instead of Code.

Hence the code is rejected, as desired.

Is this idea enough?

Consider the following code:

$$\begin{aligned} \mathsf{l} : \quad & \mathsf{r1} := \mathsf{5}; \\ & \mathsf{r2} := \mathsf{l}'; \\ & \mathsf{jump}\ \mathsf{r2} \end{aligned}$$

Label l' points to some other instruction sequence I'.

$I = \mathsf{r1} := \mathsf{5};\ \mathsf{jump}\ \mathsf{r1}$ and heap $H = \{\mathsf{l} : I, \mathsf{l}' \mapsto I'\}$.

Should the above code be well-typed? After the first two instructions, the register file type will be $\{\mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Code}\}$, as it should be.

Answer: depends on I'...

Consider the code

l' : jump r1;

Clearly the instruction sequence $I' =$ jump r1 expects a label in r1 instead of an integer.

Hence the code at l is not well-typed.

Solution:

With each instruction sequence, associate a register file type that is expected at the beginning of that instruction sequence.

Secondly, enrich the notion of types. Instead of having a simple type Code for labels, we have types of the form $\mathsf{Code}(\Gamma)$ where Γ is a register file type.

We further choose a type Top which is the super type of all types.

In the previous example, the instruction sequence I' will have type

$$\{\mathsf{r1} : \mathsf{Code}\{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\}\}$$

The instruction sequence I' expects $\mathsf{r1}$ to contain label to some instruction sequence (I) which expects both registers to contain "anything".

The instruction sequence I has type $\{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\}$.

After executing the first two instructions of I, the register file type becomes $\{r1 : \mathsf{Int}, \mathsf{r2} : \mathsf{Code}\{\ldots\}\}$.

Hence the `jump` instruction doesn't type check.

The TAL-0 type system

τ ::=	operand types
Int	integers
$\mathsf{Code}(\Gamma)$	labels
Top	"any" type

The TAL-0 type system

τ ::=	operand types
Int	integers
$\mathsf{Code}(\Gamma)$	labels
Top	"any" type

Γ ::=	register file types
$\{\mathsf{r1} : \tau_1, \ldots, \mathsf{rk} : \tau_k\}$	
Ψ ::=	heap types
$\{l_1 : \tau_1, \ldots, l_m : \tau_m\}$	

The TAL-0 type system

τ ::=		operand types	Γ ::=		register file types
	Int	integers		$\{\mathsf{r1} : \tau_1, \ldots, \mathsf{rk} : \tau_k\}$	
	Code(Γ)	labels	Ψ ::=		heap types
	Top	"any" type		$\{l_1 : \tau_1, \ldots, l_m : \tau_m\}$	

Typing of operands

The type judgment

$$\Psi, \Gamma \vdash \nu : \tau$$

means: under heap type Ψ and register file type Γ, the operand ν has type τ.

The TAL-0 type system

$\tau ::=$	operand types	$\Gamma ::=$	register file types
Int	integers		$\{\mathsf{r1} : \tau_1, \ldots, \mathsf{rk} : \tau_k\}$
$\mathsf{Code}(\Gamma)$	labels	$\Psi ::=$	heap types
Top	"any" type		$\{l_1 : \tau_1, \ldots, l_m : \tau_m\}$

Typing of operands

The type judgment

$$\Psi, \Gamma \vdash \nu : \tau$$

means: under heap type Ψ and register file type Γ, the operand ν has type τ.

$$\Psi, \Gamma \vdash n : \mathsf{Int} \qquad \text{(T-Int)}$$

The TAL-0 type system

$\tau ::=$ operand types

Int integers

$\mathsf{Code}(\Gamma)$ labels

Top "any" type

$\Gamma ::=$ register file types

$\{\mathsf{r1} : \tau_1, \ldots, \mathsf{rk} : \tau_k\}$

$\Psi ::=$ heap types

$\{l_1 : \tau_1, \ldots, l_m : \tau_m\}$

Typing of operands

The type judgment

$$\Psi, \Gamma \vdash \nu : \tau$$

means: under heap type Ψ and register file type Γ, the operand ν has type τ.

$$\Psi, \Gamma \vdash n : \mathsf{Int} \quad \text{(T-Int)} \qquad \frac{l : \tau \in \Psi}{\Psi, \Gamma \vdash l : \tau} \quad \text{(T-Lab)}$$

$$\Psi, \Gamma \vdash r : \Gamma(r) \qquad \text{(T-Reg)}$$

$$\Psi, \Gamma \vdash r : \Gamma(r) \qquad \text{(T-Reg)}$$

$$\frac{\Psi, \Gamma \vdash \nu : \tau \qquad \tau' \sqsubseteq \tau}{\Psi, \Gamma \vdash \nu : \tau'} \ \text{(T-Sub)}$$

$$\Psi, \Gamma \vdash r : \Gamma(r) \qquad \text{(T-Reg)}$$

$$\frac{\Psi, \Gamma \vdash \nu : \tau \qquad \tau' \sqsubseteq \tau}{\Psi, \Gamma \vdash \nu : \tau'} \text{(T-Sub)}$$

where

$$\begin{array}{rll}
\tau & \sqsubseteq_1 \tau & \text{for every } \tau \\
\tau & \sqsubseteq_1 \mathsf{Top} & \text{for every } \tau \\
\mathsf{Code}(\Gamma_1) & \sqsubseteq \mathsf{Code}(\Gamma_2) & \text{iff } \Gamma_1(r) \sqsubseteq_1 \Gamma_2(r) \text{ for every register } r
\end{array}$$

Top represents "any" type, hence can be replaced by any type.

Typing of instructions

The type judgment

$$\Psi \vdash \iota : \Gamma_1 \to \Gamma_2$$

means: under heap type Ψ, the instruction ι modifies the register file type from Γ_1 to Γ_2.

Typing of instructions

The type judgment

$$\Psi \vdash \iota : \Gamma_1 \to \Gamma_2$$

means: under heap type Ψ, the instruction ι modifies the register file type from Γ_1 to Γ_2.

$$\frac{\Psi, \Gamma \vdash \nu : \tau}{\Psi \vdash r_d := \nu : \Gamma \to \Gamma \oplus \{r_d : \tau\}} \text{ (T-Mov)}$$

Typing of instructions

The type judgment

$$\Psi \vdash \iota : \Gamma_1 \rightarrow \Gamma_2$$

means: under heap type Ψ, the instruction ι modifies the register file type from Γ_1 to Γ_2.

$$\frac{\Psi, \Gamma \vdash \nu : \tau}{\Psi \vdash r_d := \nu : \Gamma \rightarrow \Gamma \oplus \{r_d : \tau\}} \text{(T-Mov)}$$

$$\frac{\Psi, \Gamma \vdash r_s : \mathsf{Int} \qquad \Psi, \Gamma \vdash \nu : \mathsf{Int}}{\Psi \vdash r_d := r_s + \nu : \Gamma \rightarrow \Gamma \oplus \{r_d : \mathsf{Int}\}} \text{(T-Add)}$$

The mov and add instructions modify the type of the destination register.

$$\frac{\Psi, \Gamma \vdash r_s : \mathsf{Int} \qquad \Psi, \Gamma \vdash \nu : \mathsf{Code}(\Gamma)}{\Psi \vdash \mathsf{if}\ r_s\ \mathsf{jump}\ \nu : \Gamma \to \Gamma} \text{(T-If)}$$

Both branches of the if instruction must have the same type.

If the if condition fails then the next instruction is executed with register file of type Γ.

If the if condition succeeds then the jump should be to some instruction sequence which expects register file type Γ.

Typing of instruction sequences

The type judgment

$$\Psi : I : \mathsf{Code}(\Gamma)$$

means: under heap type Ψ, the instruction sequence I expects the register file to have type Γ at the beginning.

Typing of instruction sequences

The type judgment

$$\Psi : I : \mathsf{Code}(\Gamma)$$

means: under heap type Ψ, the instruction sequence I expects the register file to have type Γ at the beginning.

$$\frac{\Psi, \Gamma \vdash \nu : \mathsf{Code}(\Gamma)}{\Psi \vdash \ \mathsf{jump}\ \nu : \mathsf{Code}(\Gamma)} \text{(T-Jump)}$$

Typing of instruction sequences

The type judgment

$$\Psi : I : \mathsf{Code}(\Gamma)$$

means: under heap type Ψ, the instruction sequence I expects the register file to have type Γ at the beginning.

$$\frac{\Psi, \Gamma \vdash \nu : \mathsf{Code}(\Gamma)}{\Psi \vdash \ \mathsf{jump}\ \nu : \mathsf{Code}(\Gamma)} \ \text{(T-Jump)}$$

$$\frac{\Psi \vdash \iota : \Gamma_1 \rightarrow \Gamma_2 \qquad \Psi \vdash I : \mathsf{Code}(\Gamma_2)}{\Psi \vdash \iota; I : \mathsf{Code}(\Gamma_1)} \ \text{(T-Seq)}$$

Typing of register files, heaps, and machine states

$$\frac{\Psi, \_ \vdash R(\mathsf{r1}) : \Gamma(\mathsf{r1}) \qquad \ldots \qquad \Psi, \_ \vdash R(\mathsf{rk}) : \Gamma(\mathsf{rk})}{\Psi \vdash R : \Gamma} \text{ (T-Regfile)}$$

$\_$ means that the register file type is irrelevant here

Typing of register files, heaps, and machine states

$$\frac{\Psi, \_ \vdash R(\mathsf{r1}) : \Gamma(\mathsf{r1}) \qquad \ldots \qquad \Psi, \_ \vdash R(\mathsf{rk}) : \Gamma(\mathsf{rk})}{\Psi \vdash R : \Gamma} \text{(T-Regfile)}$$

_ means that the register file type is irrelevant here

$$\frac{\forall l \in dom(\Psi) \cdot \Psi \vdash H(l) : \Psi(l)}{\vdash H : \Psi} \text{(T-Heap)}$$

$dom(\Psi)$ is the set of labels in the domain of Ψ

Typing of register files, heaps, and machine states

$$\frac{\Psi, \_ \vdash R(\mathsf{r1}) : \Gamma(\mathsf{r1}) \qquad \ldots \qquad \Psi, \_ \vdash R(\mathsf{rk}) : \Gamma(\mathsf{rk})}{\Psi \vdash R : \Gamma} \text{(T-Regfile)}$$

_ means that the register file type is irrelevant here

$$\frac{\forall l \in dom(\Psi) \cdot \Psi \vdash H(l) : \Psi(l)}{\vdash H : \Psi} \text{(T-Heap)}$$

$dom(\Psi)$ is the set of labels in the domain of Ψ

$$\frac{\vdash H : \Psi \qquad \Psi \vdash R : \Gamma \qquad \Psi \vdash I : \mathsf{Code}(\Gamma)}{\vdash (H, R, I)} \text{(T-Mach)}$$

The last judgment means that (H, R, I) is a well-typed machine.

Example

$$\mathsf{l} : \underbrace{\mathsf{r1} := \mathsf{l}; \mathsf{r2} := \mathsf{l}';\ \mathsf{jump}\ \mathsf{r2}}_{I} \qquad\qquad \mathsf{l}' : \underbrace{\mathsf{jump}\ \mathsf{r1}}_{I'}$$

We have the heap $H = \{\mathsf{l} \mapsto I, \mathsf{l}' \mapsto I'\}$.

Define heap type $\Psi = \left\{ \begin{array}{l} \mathsf{l} : \mathsf{Code}\{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\}, \\ \mathsf{l}' : \mathsf{Code}\{\mathsf{r1} : \Psi(\mathsf{l}), \mathsf{r2} : \mathsf{Top}\} \end{array} \right\}$

Define register file types $\begin{array}{l} \Gamma_1 = \{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\} \\ \Gamma_2 = \{\mathsf{r1} : \Psi(\mathsf{l}), \mathsf{r2} : \mathsf{Top}\} \\ \Gamma_3 = \{\mathsf{r1} : \Psi(\mathsf{l}), \mathsf{r2} : \Psi(\mathsf{l}')\} \end{array}$

$\boxed{\text{claim 1: } \Psi \vdash I : \mathsf{Code}(\Gamma_1)}$

$$\boxed{\text{claim 1: } \Psi \vdash I : \mathsf{Code}(\Gamma_1)}$$

$$\cfrac{\cfrac{\mathsf{l} : \mathsf{Code}\{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\} \in \Psi}{\Psi, \Gamma_1 \vdash \mathsf{l} : \Psi(\mathsf{l})}\,(\text{T-Lab})}{\Psi \vdash \mathsf{r1} := \mathsf{l} : \Gamma_1 \to \Gamma_2}\,(\text{T-Mov})$$

$\boxed{\text{claim 1: } \Psi \vdash I : \mathsf{Code}(\Gamma_1)}$

$$\cfrac{\cfrac{\mathsf{l} : \mathsf{Code}\{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\} \in \Psi}{\Psi, \Gamma_1 \vdash \mathsf{l} : \Psi(\mathsf{l})}\,(\text{T-Lab})}{\Psi \vdash \mathsf{r1} := \mathsf{l} : \Gamma_1 \to \Gamma_2}\,(\text{T-Mov}) \qquad \begin{array}{c} \vdots \\ \Psi \vdash \mathsf{r2} := \mathsf{l}' : \Gamma_2 \to \Gamma_3 \end{array}$$

$$\boxed{\text{claim 1: } \Psi \vdash I : \mathsf{Code}(\Gamma_1)}$$

$$\dfrac{\dfrac{\mathsf{l} : \mathsf{Code}\{\mathsf{r1} : \mathsf{Top}, \mathsf{r2} : \mathsf{Top}\} \in \Psi}{\Psi, \Gamma_1 \vdash \mathsf{l} : \Psi(\mathsf{l})}\ (\text{T-Lab})}{\Psi \vdash \mathsf{r1} := \mathsf{l} : \Gamma_1 \to \Gamma_2}\ (\text{T-Mov}) \qquad \begin{array}{c} \vdots \\ \Psi \vdash \mathsf{r2} := \mathsf{l}' : \Gamma_2 \to \Gamma_3 \end{array}$$

$$\dfrac{\dfrac{\Psi, \Gamma_3 \vdash \mathsf{r2} : \Psi(\mathsf{l}') \quad \mathsf{Code}(\Gamma_3) \sqsubseteq \Psi(\mathsf{l}')}{\Psi, \Gamma_3 \vdash \mathsf{r2} : \mathsf{Code}(\Gamma_3)}\ (\text{T-Sub})}{\Psi \vdash \ \mathsf{jump}\ \mathsf{r2} : \mathsf{Code}(\Gamma_3)}\ (\text{T-Jump})$$

$$\begin{aligned} \mathsf{Code}(\Gamma_3) &= \mathsf{Code}\{\mathsf{r1} : \Psi(\mathsf{l}), \quad \mathsf{r2} : \Psi(\mathsf{l}')\} \\ \sqsubseteq \Psi(\mathsf{l}') &= \mathsf{Code}\{\mathsf{r1} : \Psi(\mathsf{l}), \quad \mathsf{r2} : \mathsf{Top}\} \end{aligned}$$

because $\Psi(\mathsf{l}) \sqsubseteq_1 \Psi(\mathsf{l})$ and $\Psi(\mathsf{l}') \sqsubseteq_1 \mathsf{Top}$.

$$\dfrac{\begin{array}{c}\vdots\\ \Psi \vdash \mathsf{r1} := \mathsf{l} : \Gamma_1 \to \Gamma_2\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \Psi : \mathsf{r2} := \mathsf{l}' : \Gamma_2 \to \Gamma_3\end{array} \qquad \begin{array}{c}\vdots\\ \Psi \vdash \mathsf{jump}\ \mathsf{r2} : \mathsf{Code}(\Gamma_3)\end{array}}{\Psi \vdash \mathsf{r2} := \mathsf{l}';\ \mathsf{jump}\ \mathsf{r2} : \mathsf{Code}(\Gamma_2)}\ (\text{T-Seq})}{\Psi \vdash I : \mathsf{Code}(\Gamma_1)}\ (\text{T-Seq})$$

This proves claim 1.

$$\dfrac{\begin{array}{c}\vdots\\ \Psi \vdash \mathsf{r1} := \mathsf{l} : \Gamma_1 \to \Gamma_2\end{array} \qquad \dfrac{\begin{array}{c}\vdots\\ \Psi : \mathsf{r2} := \mathsf{l}' : \Gamma_2 \to \Gamma_3\end{array} \qquad \begin{array}{c}\vdots\\ \Psi \vdash \mathsf{jump}\ \mathsf{r2} : \mathsf{Code}(\Gamma_3)\end{array}}{\Psi \vdash \mathsf{r2} := \mathsf{l}';\ \mathsf{jump}\ \mathsf{r2} : \mathsf{Code}(\Gamma_2)}\,(\text{T-Seq})}{\Psi \vdash I : \mathsf{Code}(\Gamma_1)}\,(\text{T-Seq})$$

This proves claim 1.

$\boxed{\text{claim 2: } \Psi \vdash I' : \mathsf{Code}(\Gamma_2)}$

$$\dfrac{\dfrac{\Psi, \Gamma_2 \vdash \mathsf{r1} : \Psi(\mathsf{l}) \qquad \mathsf{Code}(\Gamma_2) \sqsubseteq \Psi(\mathsf{l})}{\Psi, \Gamma_2 \vdash \mathsf{r1} : \mathsf{Code}(\Gamma_2)}\,(\text{T-Sub})}{\Psi \vdash \mathsf{jump}\ \mathsf{r1} : \mathsf{Code}(\Gamma_2)}\,(\text{T-Jump})$$

Well typing of the heap

Recall that $H = \{\mathsf{l} \mapsto I, \mathsf{l}' \mapsto I'\}$ and $\Psi = \{\mathsf{l} : \mathsf{Code}(\Gamma_1), \mathsf{l}' : \mathsf{Code}(\Gamma_2)\}$.

$$\dfrac{\begin{array}{c}\vdots\\ \Psi \vdash I : \mathsf{Code}(\Gamma_1)\end{array} \qquad \begin{array}{c}\vdots\\ \Psi \vdash I' : \mathsf{Code}(\Gamma_2)\end{array}}{\vdash H : \Psi}\ \text{(T-Heap)}$$

Well typing of the heap

Recall that $H = \{\mathsf{l} \mapsto I, \mathsf{l}' \mapsto I'\}$ and $\Psi = \{\mathsf{l} : \mathsf{Code}(\Gamma_1), \mathsf{l}' : \mathsf{Code}(\Gamma_2)\}$.

$$\frac{\begin{array}{c}\vdots\\ \Psi \vdash I : \mathsf{Code}(\Gamma_1)\end{array} \qquad \begin{array}{c}\vdots\\ \Psi \vdash I' : \mathsf{Code}(\Gamma_2)\end{array}}{\vdash H : \Psi}\ \text{(T-Heap)}$$

Well typing of register file

Suppose we want to start running the machine with the register file

$$R = \{\mathsf{r1} \mapsto \mathsf{0}, \mathsf{r2} \mapsto \mathsf{0}\}$$

Well typing of the heap

Recall that $H = \{\mathsf{l} \mapsto I, \mathsf{l}' \mapsto I'\}$ and $\Psi = \{\mathsf{l} : \mathsf{Code}(\Gamma_1), \mathsf{l}' : \mathsf{Code}(\Gamma_2)\}$.

$$\frac{\begin{array}{c}\vdots\\ \Psi \vdash I : \mathsf{Code}(\Gamma_1)\end{array} \qquad \begin{array}{c}\vdots\\ \Psi \vdash I' : \mathsf{Code}(\Gamma_2)\end{array}}{\vdash H : \Psi} \text{ (T-Heap)}$$

Well typing of register file

Suppose we want to start running the machine with the register file

$$R = \{\mathsf{r1} \mapsto \mathsf{0}, \mathsf{r2} \mapsto \mathsf{0}\}$$

Define register file type $\quad \Gamma = \{\mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Int}\}$

Well typing of the heap

Recall that $H = \{\mathsf{l} \mapsto I, \mathsf{l}' \mapsto I'\}$ and $\Psi = \{\mathsf{l} : \mathsf{Code}(\Gamma_1), \mathsf{l}' : \mathsf{Code}(\Gamma_2)\}$.

$$\frac{\begin{array}{c}\vdots \\ \Psi \vdash I : \mathsf{Code}(\Gamma_1)\end{array} \qquad \begin{array}{c}\vdots \\ \Psi \vdash I' : \mathsf{Code}(\Gamma_2)\end{array}}{\vdash H : \Psi}\ (\text{T-Heap})$$

Well typing of register file

Suppose we want to start running the machine with the register file

$$R = \{\mathsf{r1} \mapsto \mathsf{0}, \mathsf{r2} \mapsto \mathsf{0}\}$$

Define register file type $\quad \Gamma = \{\mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Int}\}$

$$\frac{\dfrac{}{\Psi, \_ \vdash \mathsf{0} : \mathsf{Int}}\ (\text{T-Int}) \qquad \dfrac{}{\Psi, \_ \vdash \mathsf{0} : \mathsf{Int}}\ (\text{T-Int})}{\Psi \vdash R : \Gamma}\ (\text{TRegfile})$$

Suppose the initial instruction sequence we want to execute is I.

We have shown that $\Psi \vdash I : \mathsf{Code}(\Gamma_1)$ (claim 1).

Similarly we show $\Psi \vdash I : \mathsf{Code}(\Gamma)$.

Suppose the initial instruction sequence we want to execute is I.

We have shown that $\Psi \vdash I : \mathsf{Code}(\Gamma_1)$ (claim 1).

Similarly we show $\Psi \vdash I : \mathsf{Code}(\Gamma)$.

Finally, well typing of the machine

$$\frac{\begin{array}{c}\vdots\\ \vdash H : \Psi\end{array} \qquad \begin{array}{c}\vdots\\ \Psi \vdash R : \Gamma\end{array} \qquad \begin{array}{c}\vdots\\ \Psi \vdash I : \mathsf{Code}(\Gamma)\end{array}}{\vdash (H, R, I)} \text{(T-Mach)}$$

Another example

```
prod :   r3 := 0;
         jump loop

loop :   if r1 jump done;
         r3 := r2 + r3;
         r1 := r1 + −1;
         jump loop

done :   jump r4
```

Another example

```
prod :  r3 := 0;
        jump loop

loop :  if r1 jump done;
        r3 := r2 + r3;
        r1 := r1 + -1;
        jump loop

done :  jump r4
```

To complete the example we will have r4 contain the halt label.

```
halt :  jump halt
```

Another example

```
prod : r3 := 0;
       jump loop

loop : if r1 jump done;
       r3 := r2 + r3;
       r1 := r1 + -1;
       jump loop

done : jump r4
```

To complete the example we will have r4 contain the halt label.

```
halt : jump halt
```

Name the instructions $\iota_1, \ldots, \iota_8$ and the instruction sequences I_1, I_2, I_3, I_4.

Let $\Gamma' = \{\mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Int}, \mathsf{r3} : \mathsf{Int}, \mathsf{r4} : \mathsf{Top}\}$

Let $\Gamma = \{\mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Int}, \mathsf{r3} : \mathsf{Int}, \mathsf{r4} : \mathsf{Code}(\Gamma')\}$

Let $H = \{\mathsf{prod} \mapsto I_1, \mathsf{loop} \mapsto I_2, \mathsf{done} \mapsto I_3, \mathsf{halt} \mapsto I_4\}$.

Let $\Psi = \{\mathsf{prod} : \mathsf{Code}(\Gamma), \mathsf{loop} : \mathsf{Code}(\Gamma), \mathsf{done} : \mathsf{Code}(\Gamma), \mathsf{halt} : \mathsf{Code}(\Gamma')\}$.

$$\dfrac{\dfrac{\dfrac{}{\Psi,\Gamma \vdash \mathsf{r3} : \mathsf{Int}}\,(\text{T-Reg}) \qquad \dfrac{}{\Psi,\Gamma \vdash \mathsf{0} : \mathsf{Int}}\,(\text{T-Int})}{\Psi \vdash \iota_1 : \Gamma \longrightarrow \Gamma}\,(\text{T-Mov}) \qquad \dfrac{\dfrac{}{\Psi,\Gamma \vdash \mathsf{loop} : \mathsf{Code}(\Gamma)}\,(\text{T-Lab})}{\Psi \vdash \mathsf{jump\ loop} : \mathsf{Code}(\Gamma)}\,(\text{T-Jump})}{\Psi \vdash I_1 : \mathsf{Code}(\Gamma)}\,(\text{T-Seq})$$

$$\dfrac{\dfrac{\dfrac{}{\Psi, \Gamma \vdash \mathsf{r3} : \mathsf{Int}}\,\text{(T-Reg)} \quad \dfrac{}{\Psi, \Gamma \vdash \mathsf{0} : \mathsf{Int}}\,\text{(T-Int)}}{\Psi \vdash \iota_1 : \Gamma \to \Gamma}\,\text{(T-Mov)} \qquad \dfrac{\dfrac{}{\Psi, \Gamma \vdash \mathsf{loop} : \mathsf{Code}(\Gamma)}\,\text{(T-Lab)}}{\Psi \vdash \mathsf{jump\ loop} : \mathsf{Code}(\Gamma)}\,\text{(T-Jump)}}{\Psi \vdash I_1 : \mathsf{Code}(\Gamma)}\,\text{(T-Seq)}$$

Similarly, $\Psi \vdash I_2 : \mathsf{Code}(\Gamma)$.

$$\dfrac{\dfrac{\dfrac{}{\Psi, \Gamma \vdash \mathsf{r3} : \mathsf{Int}}\,(\text{T-Reg}) \qquad \dfrac{}{\Psi, \Gamma \vdash \mathsf{0} : \mathsf{Int}}\,(\text{T-Int})}{\Psi \vdash \iota_1 : \Gamma \rightarrow \Gamma}\,(\text{T-Mov}) \qquad \dfrac{\dfrac{}{\Psi, \Gamma \vdash \mathsf{loop} : \mathsf{Code}(\Gamma)}\,(\text{T-Lab})}{\Psi \vdash \mathsf{jump\ loop} : \mathsf{Code}(\Gamma)}\,(\text{T-Jump})}{\Psi \vdash I_1 : \mathsf{Code}(\Gamma)}\,(\text{T-Seq})$$

Similarly, $\Psi \vdash I_2 : \mathsf{Code}(\Gamma)$.

$$\dfrac{\dfrac{\dfrac{}{\Psi, \Gamma \vdash \mathsf{r4} : \mathsf{Code}\{\mathsf{r1} : \mathsf{Int}, \mathsf{r2} : \mathsf{Int}, \mathsf{r3} : \mathsf{Int}, \mathsf{r4} : \mathsf{Top}\}}\,(\text{T-Reg})}{\Psi, \Gamma \vdash \mathsf{r4} : \mathsf{Code}(\Gamma)}\,(\text{T-Sub})}{\Psi \vdash I_3 : \mathsf{Code}(\Gamma)}\,(\text{T-Jump})$$

$$\dfrac{\dfrac{\dfrac{}{\Psi,\Gamma\vdash \mathsf{r3}:\mathsf{Int}}\,(\text{T-Reg})\qquad \dfrac{}{\Psi,\Gamma\vdash \mathsf{0}:\mathsf{Int}}\,(\text{T-Int})}{\Psi\vdash \iota_1:\Gamma\to\Gamma}\,(\text{T-Mov})\qquad \dfrac{\dfrac{}{\Psi,\Gamma\vdash \mathsf{loop}:\mathsf{Code}(\Gamma)}\,(\text{T-Lab})}{\Psi\vdash\ \mathsf{jump\ loop}:\mathsf{Code}(\Gamma)}\,(\text{T-Jump})}{\Psi\vdash I_1:\mathsf{Code}(\Gamma)}\,(\text{T-Seq})$$

Similarly, $\Psi\vdash I_2:\mathsf{Code}(\Gamma)$.

$$\dfrac{\dfrac{\dfrac{}{\Psi,\Gamma\vdash \mathsf{r4}:\mathsf{Code}\{\mathsf{r1}:\mathsf{Int},\mathsf{r2}:\mathsf{Int},\mathsf{r3}:\mathsf{Int},\mathsf{r4}:\mathsf{Top}\}}\,(\text{T-Reg})}{\Psi,\Gamma\vdash \mathsf{r4}:\mathsf{Code}(\Gamma)}\,(\text{T-Sub})}{\Psi\vdash I_3:\mathsf{Code}(\Gamma)}\,(\text{T-Jump})$$

$$\dfrac{\dfrac{}{\Psi,\Gamma'\vdash \mathsf{halt}:\mathsf{Code}(\Gamma')}\,(\text{T-Lab})}{\Psi\vdash I_4:\mathsf{Code}(\Gamma')}\,(\text{T-Jump})$$

Hence we have well typing of the machine:

$$\dfrac{\begin{array}{cccc} \vdots & \vdots & \vdots & \vdots \\ I_1 : \mathsf{Code}(\Gamma) & I_2 : \mathsf{Code}(\Gamma) & I_3 : \mathsf{Code}(\Gamma) & I_4 : \mathsf{Code}(\Gamma') \end{array}}{\vdash H : \Psi}\ \text{(T-Heap)}$$

Hence we have well typing of the machine:

$$\dfrac{\begin{array}{c}\vdots\\ I_1 : \mathsf{Code}(\Gamma)\end{array} \qquad \begin{array}{c}\vdots\\ I_2 : \mathsf{Code}(\Gamma)\end{array} \qquad \begin{array}{c}\vdots\\ I_3 : \mathsf{Code}(\Gamma)\end{array} \qquad \begin{array}{c}\vdots\\ I_4 : \mathsf{Code}(\Gamma')\end{array}}{\vdash H : \Psi}\ (\text{T-Heap})$$

Define initial register file: $R = \{\mathsf{r1} \mapsto \mathsf{0}, \mathsf{r2} \mapsto \mathsf{0}, \mathsf{r3} \mapsto \mathsf{0}, \mathsf{r4} \mapsto \mathsf{halt}\}$

$$\dfrac{\dfrac{}{\Psi, \_ \vdash \mathsf{0} : \mathsf{Int}}\ (\text{T-Int}) \qquad \ldots \qquad \dfrac{}{\Psi, \_ \vdash \mathsf{0} : \mathsf{Int}}\ (\text{T-Int}) \qquad \dfrac{}{\Psi, \_ \vdash \mathsf{halt} : \mathsf{Code}(\Gamma')}\ (\text{T-Int})}{\Psi \vdash R : \Gamma}\ (\text{T-Regfile})$$

Hence we have well typing of the machine:

$$\dfrac{\begin{array}{c}\vdots\\ I_1 : \mathsf{Code}(\Gamma)\end{array} \quad \begin{array}{c}\vdots\\ I_2 : \mathsf{Code}(\Gamma)\end{array} \quad \begin{array}{c}\vdots\\ I_3 : \mathsf{Code}(\Gamma)\end{array} \quad \begin{array}{c}\vdots\\ I_4 : \mathsf{Code}(\Gamma')\end{array}}{\vdash H : \Psi}\ (\text{T-Heap})$$

Define initial register file: $R = \{\mathsf{r1} \mapsto \mathsf{0}, \mathsf{r2} \mapsto \mathsf{0}, \mathsf{r3} \mapsto \mathsf{0}, \mathsf{r4} \mapsto \mathsf{halt}\}$

$$\dfrac{\dfrac{}{\Psi, \_ \vdash \mathsf{0} : \mathsf{Int}}\ (\text{T-Int}) \qquad \ldots \qquad \dfrac{}{\Psi, \_ \vdash \mathsf{0} : \mathsf{Int}}\ (\text{T-Int}) \qquad \dfrac{}{\Psi, \_ \vdash \mathsf{halt} : \mathsf{Code}(\Gamma')}\ (\text{T-Int})}{\Psi \vdash R : \Gamma}\ (\text{T-Regfile})$$

$$\dfrac{\vdash H : \Psi \qquad \Psi \vdash R : \Gamma \qquad \Psi \vdash I_1 : \mathsf{Code}(\Gamma)}{\vdash (H, R, I_1)}\ (\text{T-Mach})$$

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 mchine 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 mchine 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 mchine 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 $\vdash M'$.

Type safety for TAL-0

"well typed machines do not get stuck"

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

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

Proof: by easy induction, case analysis…

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 $\vdash M'$.

Proof: by easy induction, case analysis. . .

Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.

Type safety for TAL-0

"well typed machines do not get stuck"

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

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

Proof: by easy induction, case analysis…

Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.

A: The definitions are almost always wrong.

— Anonymous

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

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

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

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.

- salloc 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 ptr(Int), ptr(Code(Γ)), ptr(ptr(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?

```
                      // 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???
```

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).

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

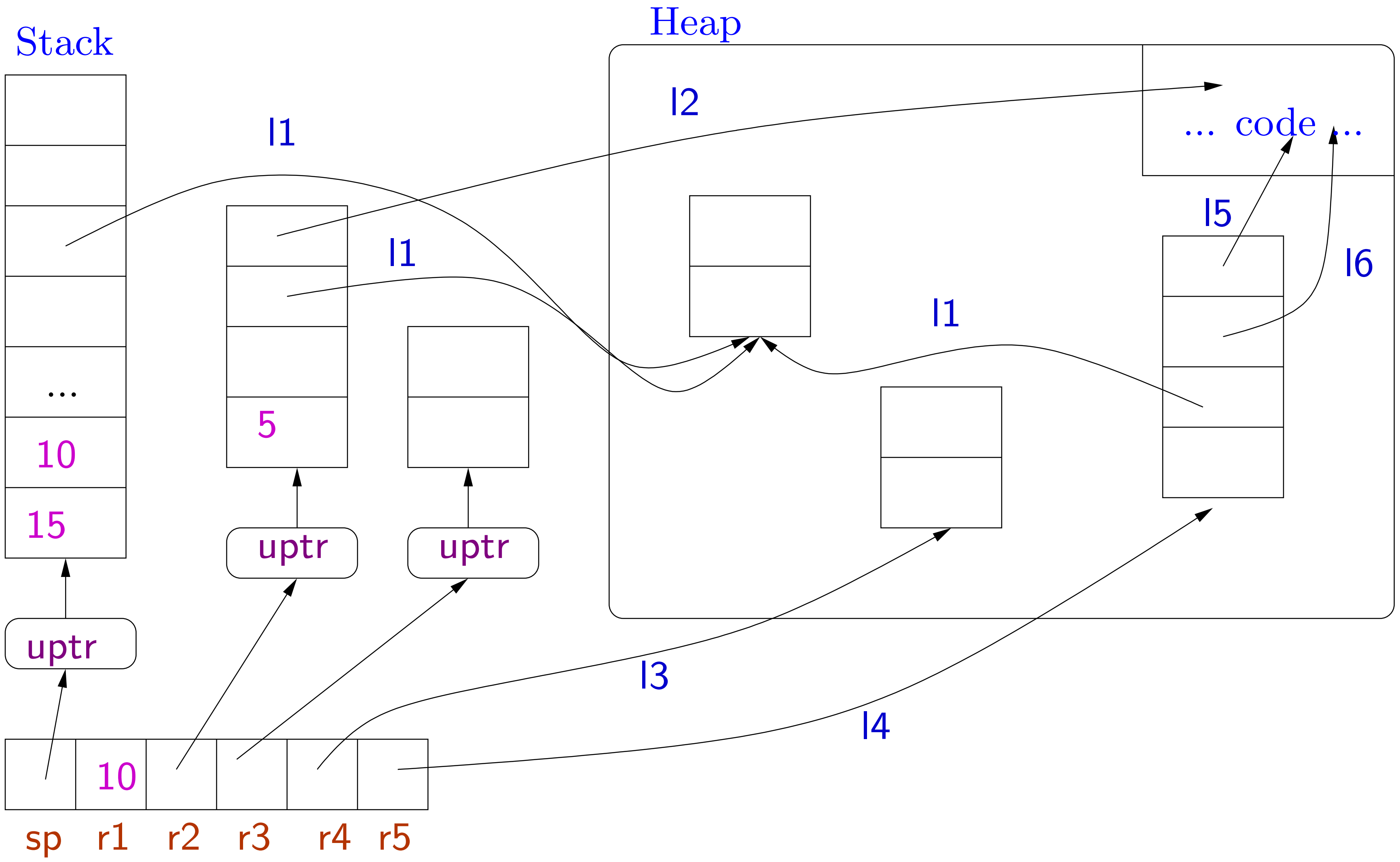
$\langle \nu_1, \ldots, \nu_n \rangle$ tuples

Instruction sequences I are 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 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 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.

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

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.

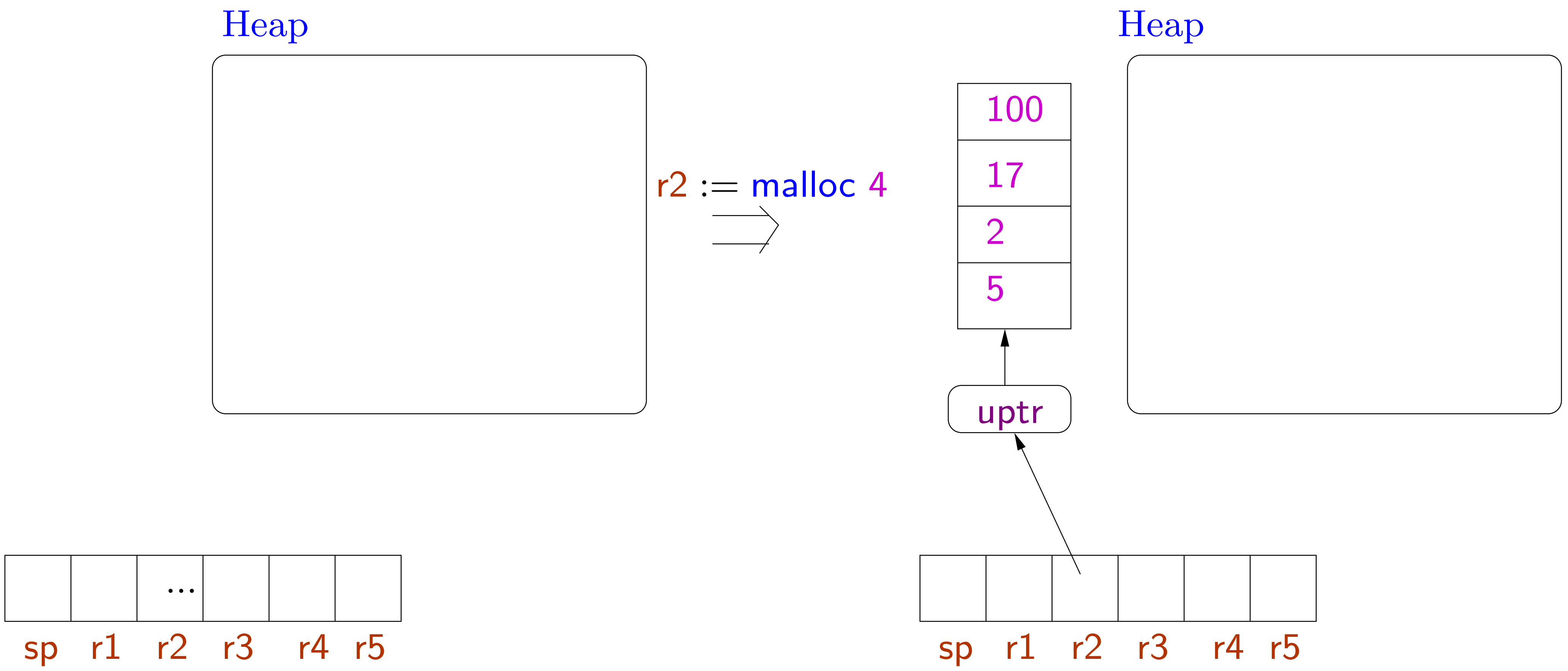
The other evaluation rules of TAL-0 are unmodified. We now add new rules for the new instructions . . .

Allocation generates a unique pointer

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

- 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 $\Longrightarrow$

Heap

100

17

2

5

uptr

…

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:

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

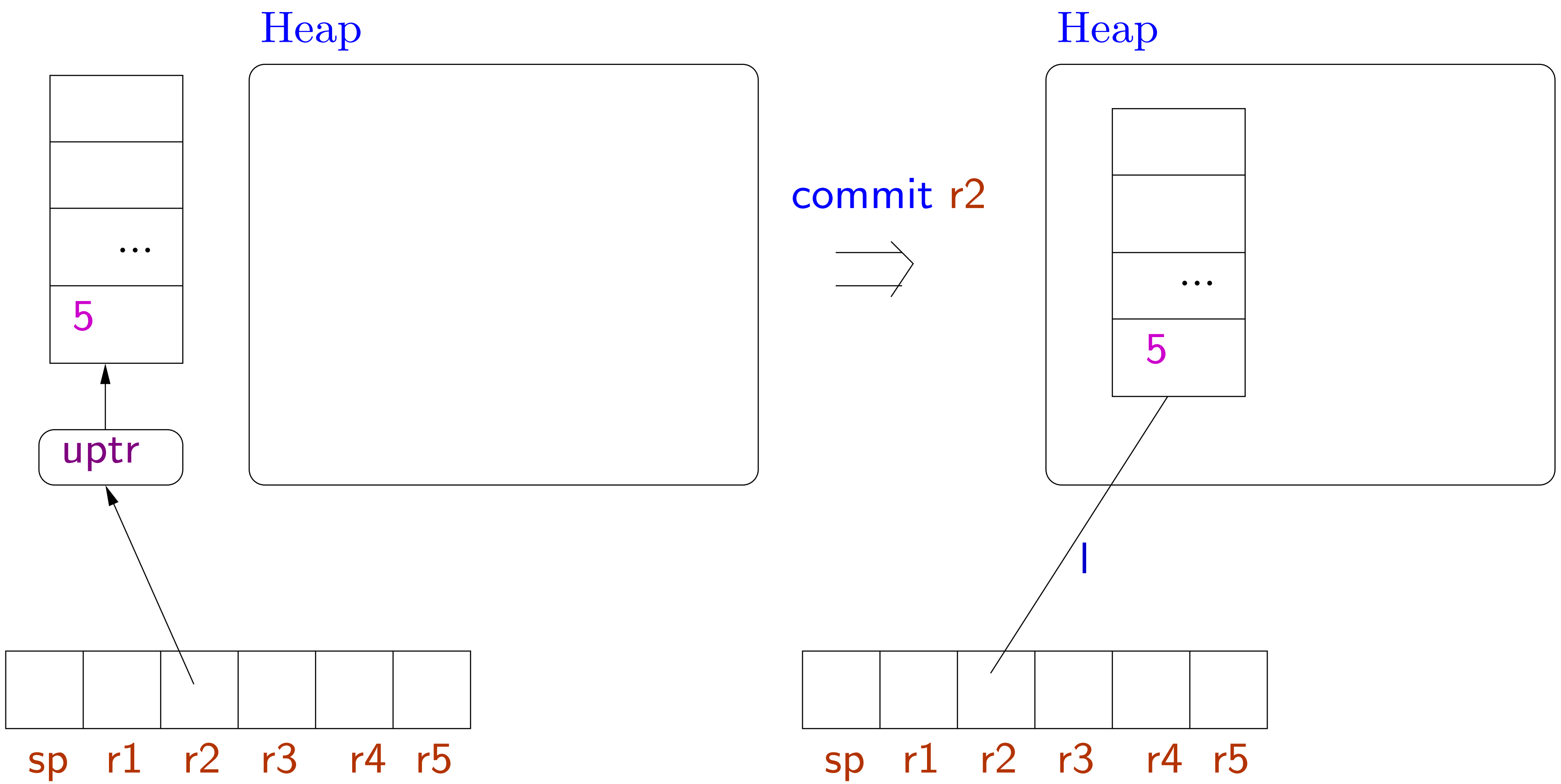
- using unique pointers in place of integers

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

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

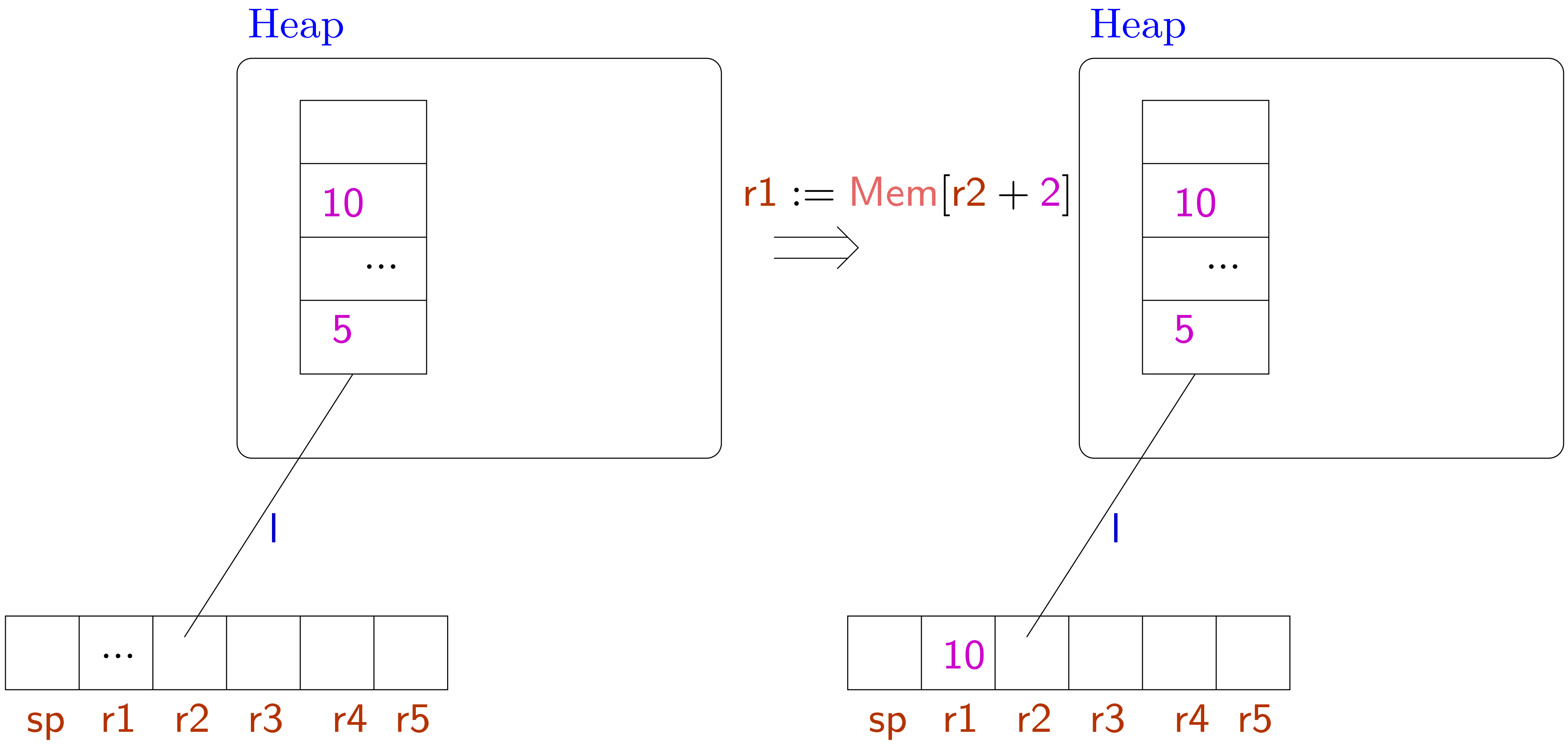
Loading shared data

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

Loading unique data

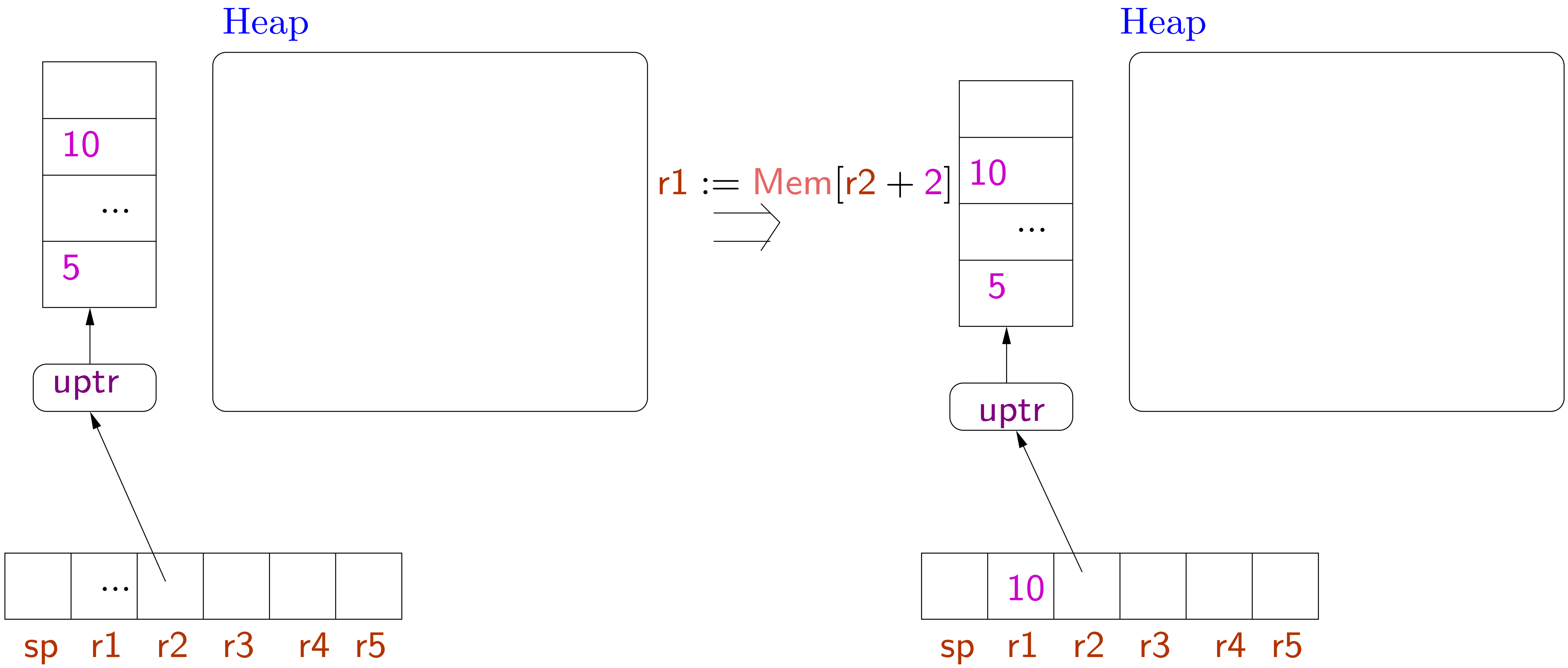
$$\frac{R(r_s) = \mathsf{uptr}\langle \nu_0, \ldots, \nu_n, \ldots, \rangle}{(H, R, r_d := \mathsf{Mem}[r_s + n]; I) \to (H, R \oplus \{r_d \mapsto \nu_n\}, I)} \text{(E-Ld-U)}$$

Loading shared data



$$\mathsf{r1} := \mathsf{Mem}[\mathsf{r2} + 2] \Longrightarrow$$

Loading unique data



$\mathsf{r1} := \mathsf{Mem}[\mathsf{r2} + 2] \Longrightarrow$

Storing shared data

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

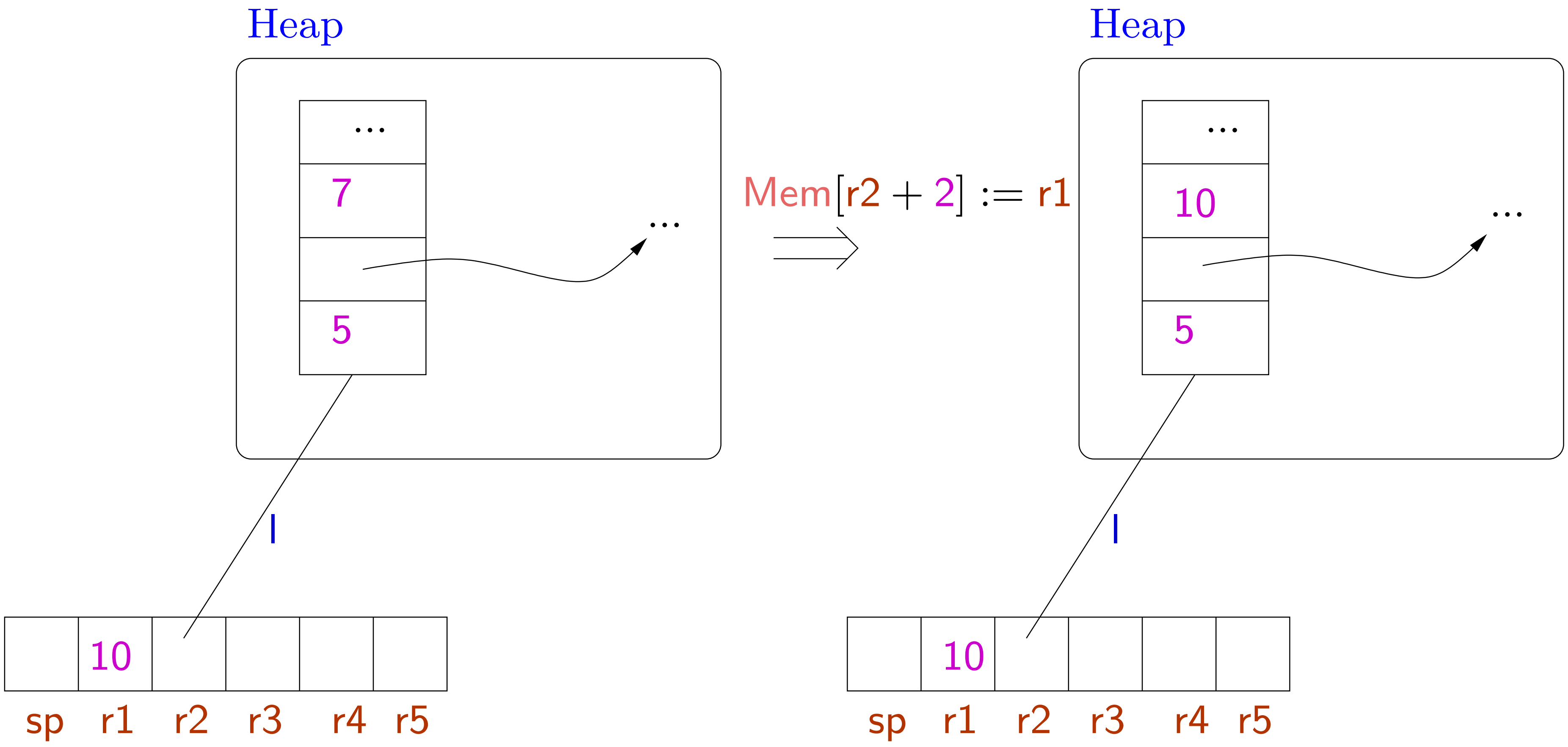
Storing shared data

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

Storing unique data

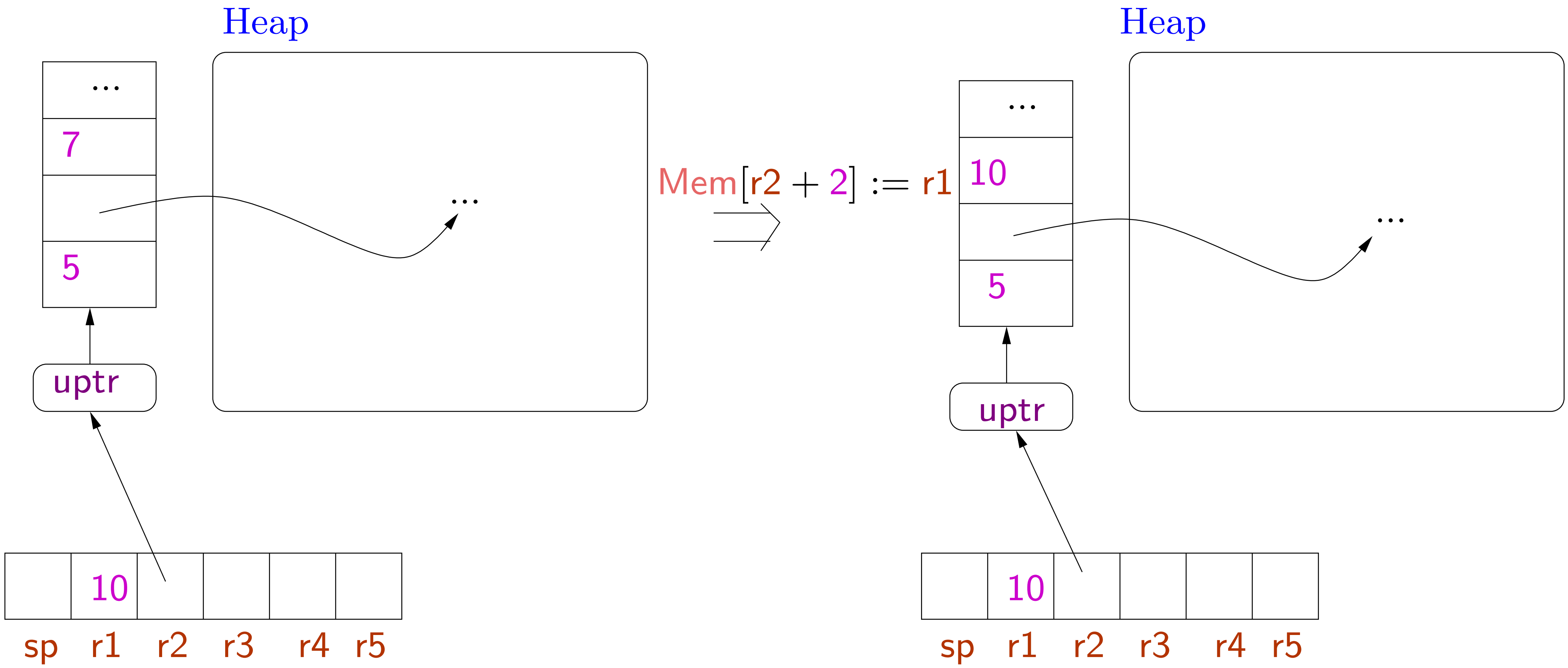
$$\frac{R(r_d) = \mathsf{uptr}\langle \nu_0, \ldots, \nu_n, \ldots, \rangle \quad R(r_s) = \nu \quad \nu \neq \mathsf{uptr}(h)}{(H, R, \mathsf{Mem}[r_d + n] := r_s; I) \rightarrow (H, R \oplus \{r_d \mapsto \mathsf{uptr}\langle \nu_0, \ldots, \nu, \ldots, \rangle\}, I)} \text{(E-St-U)}$$

Storing shared data



$\mathsf{Mem}[\mathsf{r2} + 2] := \mathsf{r1}$

Storing unique data



$\text{Mem}[r2 + 2] := r1$

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

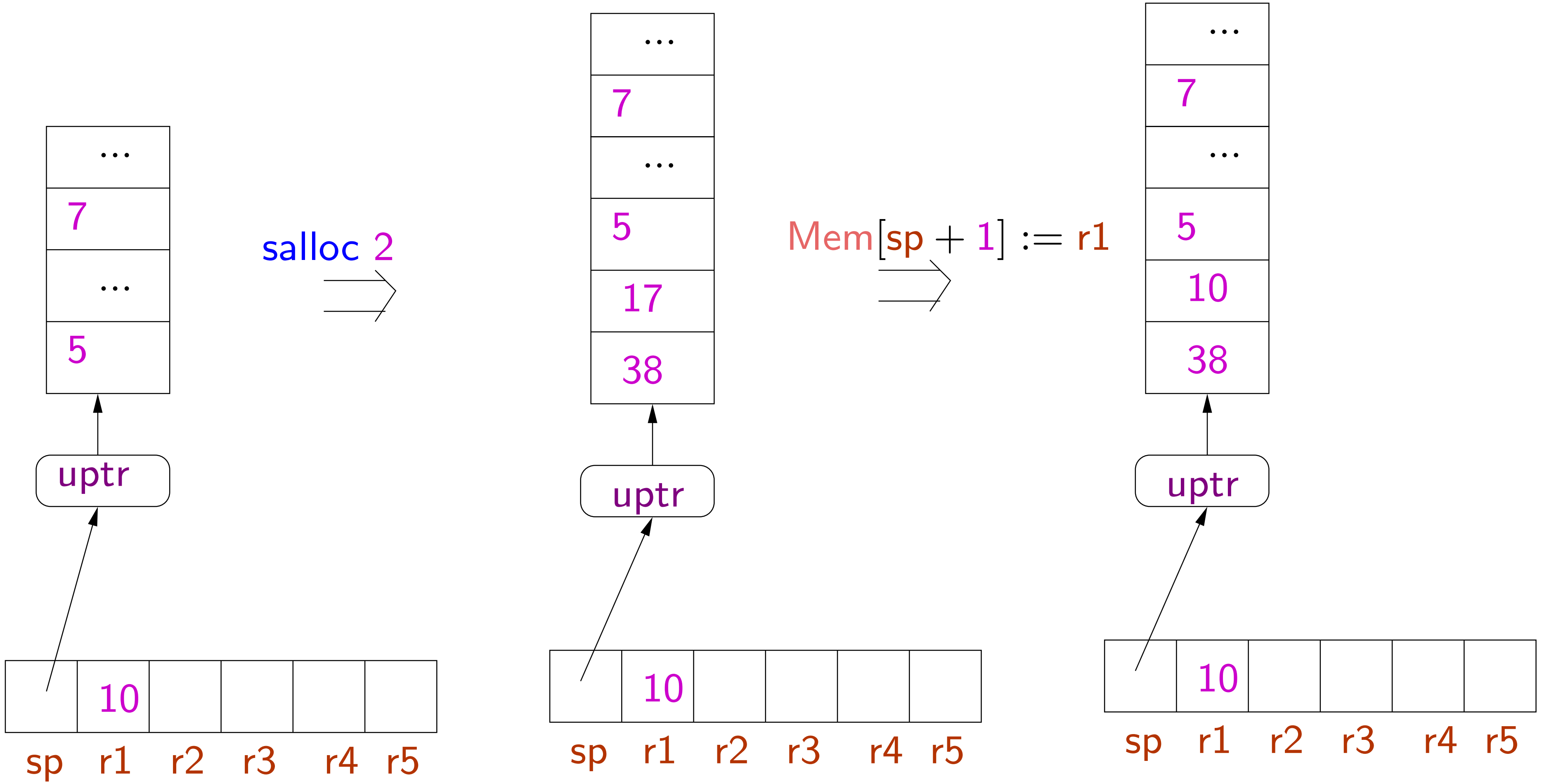
$$\frac{R(\mathsf{sp}) = \mathsf{uptr}\langle \nu_0, \ldots, \nu_p \rangle \quad p + n \leq \mathsf{MaxStack}}{(H, R, \mathsf{salloc}\ n; I) \rightarrow (H, R \oplus \{\mathsf{sp} \mapsto \mathsf{uptr}\langle m_1, \ldots, m_n, \nu_0, \ldots, \nu_p \rangle\}, I)}\ (\text{E-Salloc})$$

- 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

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

- 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-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 $\mathsf{r1} := \mathsf{malloc}\ 3$ 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.

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.

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

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.

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

We add new typing rules for the new instructions.

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

malloc creates a unique pointer type.

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

commit creates a shared pointer type.

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

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

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

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

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

Updating shared data should not involve a change in type.

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

Updating shared data should not involve a change in type.

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

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

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

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

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

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

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.

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

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.

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

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

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

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

Type Safety for TAL-1

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

Preservation: If $\vdash M$ and $M \to M'$ then either M' is $\mathsf{StackOverflow}$, or $\vdash M'$.

The Java Security Manager

Allows or disallows various operations.

Various kinds of operations (reading or writing files, connecting to another machine) requires asking the security manager for permission.

Security managers are objects of the `SecurityManager` class.

```
public class BadClass {
  public static void main(String args[]) {
    try {
      Runtime.getRuntime().exec (”/bin/rm /path/to/filexyz”);
    } catch (Exception e) {
      System.out.println (”Deletion command failed: ” + e);
      return;
    }
    System.out.println (”Deletion command successful!”);
  }
}
```

```
public class BadClass {
  public static void main(String args[]) {
    try {
      Runtime.getRuntime().exec (”/bin/rm /path/to/filexyz”);
    } catch (Exception e) {
      System.out.println (”Deletion command failed: ” + e);
      return;
    }
    System.out.println (”Deletion command successful!”);
  }
}
```

```
Deletion command successful!
```

The local file gets deleted, if the user has permissions from the operating system.

What if such code is present in some applet loaded by a web-browser?

What if such code is present in some applet loaded by a web-browser?

```
import java.applet.Applet; import java.awt.Graphics;

public class BadApplet extends Applet{

  String text;

  public void init() {
    try { Runtime.getRuntime().exec(”/bin/rm −rf /path/to/filexyz”);
    } catch (Exception e) { text = ”Deletion command failed: ” + e; return; }
    text = ”Deletion command successful!”;
  }

  public void paint(Graphics g){ g.drawString(text, 15, 25); }
}
```

This applet is used in the following HTML page.

```
<html><body>
<applet code=”BadApplet.class” width=750 HEIGHT=50></applet>
</body></html>
```

This applet is used in the following HTML page.

```
<html><body>
<applet code=”BadApplet.class” width=750 HEIGHT=50></applet>
</body></html>
```

Loading this page in a web browser shows:

```
Deletion command failed: java.security.AccessControlException:
access denied (java.io.FilePermission /bin/rm execute)
```

This applet is used in the following HTML page.

```
<html><body>
<applet code=”BadApplet.class” width=750 HEIGHT=50></applet>
</body></html>
```

Loading this page in a web browser shows:

```
Deletion command failed: java.security.AccessControlException:
access denied (java.io.FilePermission /bin/rm execute)
```

The web browser automatically give restricted permissions to applets.

The sandbox associated with a class depends upon the source from where it was loaded.

The typical sequence used for potentially dangerous operations:

- User program makes some request to the Java API.
- The Java API asks the security manager for permissions.
- If the security manager doesn't want to allow this operation, it throws back an exception which is thrown back to the user program.
- Otherwise the security manager does nothing and the Java API completes the operation.

In the previous example, the user program calls the `exec` method, which calls the `checkExec` method on the security manager to check for permission.

The code executed on calling `exec` is similar to this:

```
public process exec (String command) throws IOException {
  ...
  SecurityManager sm = System.getSecurityManager();
  if (sm != null) {
    sm.checkExec();
    // security exception can be raised here
  }
  // remaining code follows
  ...
}
```

Another example: reading files.

```
// open a file
FileInputStream fis = new FileInputStream (”somefile”);
// read a byte
int x = fis.read();
```

The code executed on calling FileInputStream is similar to

```
public FileInputStream (String name) throws FileNotFoundException {
  SecurityManager sm = System.getSecurityManager();
  if (sm != null) { sm.checkRead(name); }
  try { open (name);
  } catch (IOException e) {
    throw new FileNotFoundException (name);
  }
}
```

The System class has various useful data and functions which are global for the whole virtual machine.

The security manager is obtained by getSecurityManager method, and null is returned if no security manager has been set.

The security manager is set by setSecurityManager method, and an exception is raised if the security manager has already been set.

Hence once the security manager has been set, it cannot be modified.

In particular, java applications can set the security manager before executing remote applets, so that these applets don't try to set their own security manager.

Defining one’s own security manager: we extend the SecurityManager class and override the functions as required.

```
public class NewSecurityManager extends SecurityManager {
  public void checkExec (String cmd) {
    // always disallow exec
    throw new SecurityException (”exec not allowed”)
  }
}
```

Modifying the BadClass to use this security manager.

```
public class NewBadClass {
  public static void main(String args[]) {
    SecurityManager sm = new NewSecurityManager();
    System.setSecurityManager(sm);
    try {
      Runtime.getRuntime().exec (”/bin/rm /path/to/filexyz”);
    } catch (Exception e) {
      System.out.println (”Deletion command failed: ” + e);
      return;
    }
    System.out.println (”Deletion command successful!”);
  }
}
```

Modifying the BadClass to use this security manager.

```
public class NewBadClass {
  public static void main(String args[]) {
    SecurityManager sm = new NewSecurityManager();
    System.setSecurityManager(sm);
    try {
      Runtime.getRuntime().exec (”/bin/rm /path/to/filexyz”);
    } catch (Exception e) {
      System.out.println (”Deletion command failed: ” + e);
      return;
    }
    System.out.println (”Deletion command successful!”);
  }
}
```

```
Deletion command failed: java.lang.SecurityException: exec not allowed
```

Examples of methods of the security manager.

- checkRead (String file): called e.g. by FileInputStream (String file).
- checkWrite (String file): called by FileOutputStream (String file).
- checkDelete (String file)

Examples of methods of the security manager.

- checkRead (String file): called e.g. by FileInputStream (String file).
- checkWrite (String file): called by FileOutputStream (String file).
- checkDelete (String file)

Note that while creating a FileInputStream object requires a checkRead call, the actual read() operations on the file input stream requires no permission.

- A trusted class can choose to deliver the FileInputStream object to an untrusted class which can then read from the file.
- It is efficient to check permissions only once.

The Access Controller

- Has functions similar to the security manager.
- Provides easy enforcement of fine grained security policies.
- The security manager works most of the time by calling the access controller.
- Implemented by the AccessController class, accessed through its static methods.

Involves the following four classes.

- The CodeSource class: represents the source from which a certain class was loaded, an an optional list of certificates which was used to sign that code.
- The Permission and Permissions classes: represent various kinds of permissions.
- The Policy class: a policy maps code source objects to permission objects. Only one policy can be associated with the JVM at any point of time, like the security manager. But the policy can be modified.
- The ProtectionDomain class: a protection domain represents all the permissions granted to a particular code source.

A permission has three properties:

- A type: what kind of permission is this?
- A name: the object that this permission talks about.
- Actions

A permission has three properties:

- A type: what kind of permission is this?
- A name: the object that this permission talks about.
- Actions

Permission objects for accessing files are members of the FilePermission class (subclass of the Permission class).

- The type is FilePermission.
- The name is the name of the file.
- Possible actions are ”read”, ”write”, ”delete” and ”execute”.

A permission has three properties:

- A type: what kind of permission is this?
- A name: the object that this permission talks about.
- Actions

Permission objects for accessing files are members of the FilePermission class (subclass of the Permission class).

- The type is FilePermission.
- The name is the name of the file.
- Possible actions are ”read”, ”write”, ”delete” and ”execute”.

Permission objects are used for requesting permissions as well as for representing granted permissions.

The security manager, on receiving the checkExec(”/bin/rm”) call, would normally construct the following permission object

```
FilePermission fp = new FilePermission (”/bin/rm”, ”execute”);
```

and then query the access controller.

```
AccessController.checkPermission (fp);
```

The security manager, on receiving the `checkExec(”/bin/rm”)` call, would normally construct the following permission object

```
FilePermission fp = new FilePermission (”/bin/rm”, ”execute”);
```

and then query the access controller.

```
AccessController.checkPermission (fp);
```

Other examples:

```
FilePermission fp1 = new FilePermission (”/bin/∗”, ”execute”);
FilePermission fp2 = new FilePermission (”/home/userx”, ”read, write”);
SocketPermission sp1 = new SocketPermission (”hostname:port”, ”connect”);
SocketPermission sp1 = new SocketPermission (”hostname:port”, ”accept, listen”);
```

Policies are specified by objects of Policy class.

It can be obtained and set using getPolicy () and setPolicy (Policy p).

Policy objects can be created by reading from a file which lists the policy rules.

Typically done at startup time:

```
java −Djava.security.manager −Djava.security.policy=<policyfilename> <class> <args>
appletviewer −J−Djava.security.policy=<policyfilename> file.html
```

The policy file have rules mapping code sources to sets of permissions.

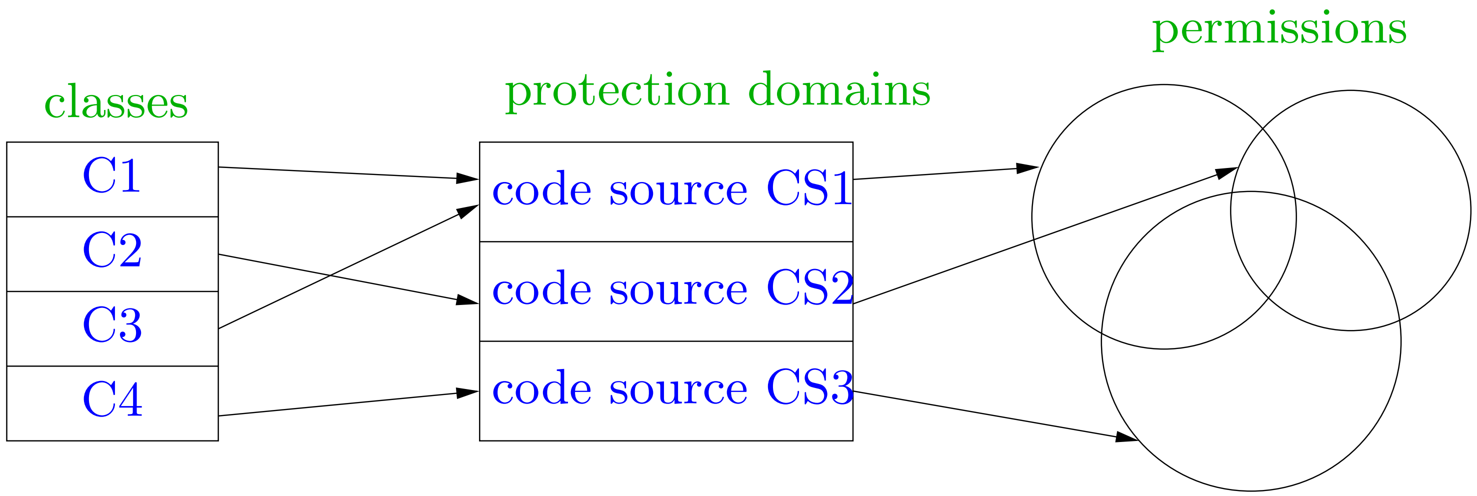
```
grant codeBase ”file:/home/userxyz/classes” {
  permission java.io.FilePermission ”/bin/rm” ”execute”;
  permission java.net.SocketPermission ”localhost:1024–” ”listen, accept”;
};

grant signedBy <signer>, codeBase ”http://www.xyz.com” {
  permission ...
  ...
};
```

A protection domain groups a code source with a set of permissions.

The class loader is supposed to associate a protection domain with a class when it loads the class.

The protection domain associated with each class is used by the access controller when it is called to check a permission using the checkPermission() method.



Stack inspection

Allowing or disallowing a permission depends on the context in which the checkPermission method was called.

The access controller needs to examine the protection domains associated with all the classes on the stack.

The permission is granted only if all the protection domains on the stack have this permission.

In our old example, the BadClass.main() method for deleting a file calls the Runtime.exec() method which calls the AccessController.checkPermission() to check execute permission on /bin/rm.

Further, the BadClass.main() method itself may be called by some other method m() of class C.

We get the following stack.

AccessController.checkPermission()
Runtime.exec()
BadClass.main()
C.m()
. . .

The execute permission should be granted only if all the classes on the stack have that permission in their protection domain.

Hence the access controller checks that all frames from the top of the stack to the bottom have this permission in the protection domains of the respective classes.

Sometimes a trusted class may choose to give its permissions to lower frames on the stack.

E.g. an untrusted applet may call some routine to draw something on the screen, and the routine requires some local font file.

This is done using the doPrivileged() method.

```
untrustedclass { f() { ... trustedclass.draw() ...}}
trustedclass {
  public void draw {
    ...
    AccessController.doPrivileged (new PrivilegedAction () {
      public Object run () {
        // privileged code here
        ... <read font file> ...
      } }); }}
```

Instead of the doPrivileged() method

```
AccessController.doPrivileged (new PrivilegedAction () {
  public Object run () {
    <privileged code>
  }
});
```

earlier versions used beginPrivileged() and endPrivileged() calls.

```
AccessController.beginPrivileged();
<privileged code>
AccessController.endPrivileged();
```

To understand the stack inspection algorithm let us assume the following operations.

- enablePrivilege(T)
- disablePrivilege(T)
- checkPrivilege(T)
- revertPrivilege(T)

where T is a target (permission in the Java terminology) we wish to protect.

Actions taken by these operations:

- enablePrivilege(T) puts an enabledPrivilege(T) flag on the current stack frame if the current class has access to T according to the policy.
- disablePrivilege(T) puts a disabledPrivilege(T) flag on the current stack frame (and removes enabledPrivilege(T) flag if present).
- revertPrivilege(T) removes enabledPrivilege(T) and disabledPrivilege(T) flags from the current stack frame if present.
- checkPrivilege(T) examines the stack as follows …

```
checkPrivilege (T) {
  for SF from top stack frame to bottom stack frame {
    if (policy doesn't allow the class in SF to access T) throw ForbiddenException;
    if (SF has enabledPrivilege (T) flag) return;
    if (SF has disabledPrivilege (T) flag) throw ForbiddedException;
  }
  return; // reached bottom of stack
}
```

The ABLP Logic

Abadi, Burrows, Lampson and Plotkin, 1993

We will model stack inspection using the (subset of) ABLP logic described below. The language contains

- Principals, modeling persons, organizations as well as cryptographic keys.
- Targets, modeling resources we wish to protect.
- Statements, modeling utterances of principals.

The ABLP Logic Abadi, Burrows, Lampson and Plotkin, 1993

We will model stack inspection using the (subset of) ABLP logic described below. The language contains

- Principals, modeling persons, organizations as well as cryptographic keys.
- Targets, modeling resources we wish to protect.
- Statements, modeling utterances of principals.
  - The statement P says $\mathsf{Ok}(T)$ means that the principal P is authorizing access to target T.

The ABLP Logic Abadi, Burrows, Lampson and Plotkin, 1993

We will model stack inspection using the (subset of) ABLP logic described below. The language contains

- Principals, modeling persons, organizations as well as cryptographic keys.
- Targets, modeling resources we wish to protect.
- Statements, modeling utterances of principals.
  - The statement P says Ok(T) means that the principal P is authorizing access to target T.
  - $P \mid Q$ says s means P says (Q says s), i.e. P quotes Q as saying s.

The ABLP Logic

Abadi, Burrows, Lampson and Plotkin, 1993

We will model stack inspection using the (subset of) ABLP logic described below. The language contains

- Principals, modeling persons, organizations as well as cryptographic keys.
- Targets, modeling resources we wish to protect.
- Statements, modeling utterances of principals.
  - The statement $P\ \mathsf{says}\ \mathsf{Ok}(T)$ means that the principal P is authorizing access to target T.
  - $P \mid Q\ \mathsf{says}\ \mathsf{s}$ means $P\ \mathsf{says}\ (Q\ \mathsf{says}\ \mathsf{s})$, i.e. P quotes Q as saying s.
  - $P \wedge Q\ \mathsf{says}\ \mathsf{s}$ means that both P and Q say s.

The ABLP Logic Abadi, Burrows, Lampson and Plotkin, 1993

We will model stack inspection using the (subset of) ABLP logic described below. The language contains

- Principals, modeling persons, organizations as well as cryptographic keys.
- Targets, modeling resources we wish to protect.
- Statements, modeling utterances of principals.
  - The statement P says Ok(T) means that the principal P is authorizing access to target T.
  - $P \mid Q$ says s means P says (Q says s), i.e. P quotes Q as saying s.
  - $P \wedge Q$ says s means that both P and Q say s.
  - $P {\Rightarrow} Q$ means that P speaks for Q, i.e. P has at least as much authority as Q.

We assume a set of atomic statements and atomic principals.

principal P ::=

$$AtomicPrincipal$$

$$P_1 \wedge P_2$$

$$P_1 \mid P_2$$

statement s ::=

$$AtomicStatement$$

$$\mathsf{s}_1 \wedge \mathsf{s}_2$$

$$\mathsf{s}_1 \longrightarrow \mathsf{s}_2$$

$$P \ \mathsf{says}\ \mathsf{s}_1$$

$$P_1 \Rightarrow P_2$$

Example Given some s we define following new statements.

$$\mathsf{s}_1 \equiv (Alice \land Bob)\ \mathsf{says}\ (Charlie {\Rightarrow} (Alice \land Bob))$$

Alice and *Bob* declare *Charlie* to be their representative.

Example Given some s we define following new statements.

$$\mathsf{s}_1 \equiv (Alice \wedge Bob)\ \mathsf{says}\ (Charlie \Rightarrow (Alice \wedge Bob))$$

$Alice$ and Bob declare $Charlie$ to be their representative.

$$\mathsf{s}_2 \equiv Charlie \mid Alice\ \mathsf{says}\ \mathsf{s}$$

$Charlie$ quotes $Alice$ as saying s.

Example Given some s we define following new statements.

$$\mathsf{s}_1 \equiv (Alice \wedge Bob)\ \mathsf{says}\ (Charlie \Rightarrow (Alice \wedge Bob))$$

Alice and *Bob* declare *Charlie* to be their representative.

$$\mathsf{s}_2 \equiv Charlie \mid Alice\ \mathsf{says}\ \mathsf{s}$$

Charlie quotes *Alice* as saying s.

$$\mathsf{s}_3 \equiv (Alice\ \mathsf{says}\ \mathsf{s}) \rightarrow \mathsf{s}$$

If *Alice* says s then it must be true.

Example Given some s we define following new statements.

$$\mathsf{s}_1 \equiv (\mathit{Alice} \wedge \mathit{Bob})\ \mathsf{says}\ (\mathit{Charlie} \Rightarrow (\mathit{Alice} \wedge \mathit{Bob}))$$

Alice and *Bob* declare *Charlie* to be their representative.

$$\mathsf{s}_2 \equiv \mathit{Charlie} \mid \mathit{Alice}\ \mathsf{says}\ \mathsf{s}$$

Charlie quotes *Alice* as saying s.

$$\mathsf{s}_3 \equiv (\mathit{Alice}\ \mathsf{says}\ \mathsf{s}) \longrightarrow \mathsf{s}$$

If *Alice* says s then it must be true.

Intuitively, from $\mathsf{s}_1 \wedge \mathsf{s}_2 \wedge \mathsf{s}_3$ we should be able to prove s.

Example Given some s we define following new statements.

$$\mathsf{s}_1 \equiv (\mathit{Alice} \wedge \mathit{Bob})\ \mathsf{says}\ (\mathit{Charlie} \Rightarrow (\mathit{Alice} \wedge \mathit{Bob}))$$

Alice and *Bob* declare *Charlie* to be their representative.

$$\mathsf{s}_2 \equiv \mathit{Charlie} \mid \mathit{Alice}\ \mathsf{says}\ \mathsf{s}$$

Charlie quotes *Alice* as saying s.

$$\mathsf{s}_3 \equiv (\mathit{Alice}\ \mathsf{says}\ \mathsf{s}) \longrightarrow \mathsf{s}$$

If *Alice* says s then it must be true.

Intuitively, from $\mathsf{s}_1 \wedge \mathsf{s}_2 \wedge \mathsf{s}_3$ we should be able to prove s.

For this we require certain rules (axioms) for making proofs.

Axioms about statements

1 *If* `s` *is an instance of a theorem of propositional logic then* `s` *is true in ABLP logic.*

Axioms about statements

1 *If* s *is an instance of a theorem of propositional logic then* s *is true in ABLP logic.*

E.g. the ABLP statement

$$(P \text{ says s}) \rightarrow (P \text{ says s})$$

is an instance of the propositional logic statement

$$X \rightarrow X$$

Axioms about statements

1 *If* s *is an instance of a theorem of propositional logic then* s *is true in ABLP logic.*

E.g. the ABLP statement

$$(P\ \mathsf{says}\ \mathsf{s}) \rightarrow (P\ \mathsf{says}\ \mathsf{s})$$

is an instance of the propositional logic statement

$$X \rightarrow X$$

The ABLP statement

$$(P\ \mathsf{says}\ \mathsf{s}) \wedge ((P\ \mathsf{says}\ \mathsf{s}) \rightarrow \mathsf{s}) \rightarrow \mathsf{s}$$

is an instance of the propositional logic statement

$$(X \wedge (X \rightarrow Y)) \rightarrow Y$$

Axioms about statements

1 *If* s *is an instance of a theorem of propositional logic then* s *is true in ABLP logic.*

E.g. the ABLP statement

$$(P \textsf{ says } \mathsf{s}) \longrightarrow (P \textsf{ says } \mathsf{s})$$

is an instance of the propositional logic statement

$$X \longrightarrow X$$

The ABLP statement

$$(P \textsf{ says } \mathsf{s}) \wedge ((P \textsf{ says } \mathsf{s}) \longrightarrow \mathsf{s}) \longrightarrow \mathsf{s}$$

is an instance of the propositional logic statement

$$(X \wedge (X \longrightarrow Y)) \longrightarrow Y$$

Hence both ABLP statements are true.

2 *If* s *and* $\mathsf{s}{\longrightarrow}\mathsf{s}'$ *then* s'.

2 *If* s *and* $\mathsf{s} \rightarrow \mathsf{s}'$ *then* s'.

3 $(P\ \mathsf{says}\ \mathsf{s} \wedge P\ \mathsf{says}\ (\mathsf{s} \rightarrow \mathsf{s}')) \rightarrow P\ \mathsf{says}\ \mathsf{s}'$

We can draw conclusions from statements made by principals.

2 *If* s *and* $\mathsf{s} \rightarrow \mathsf{s}'$ *then* s'.

3 $(P\ \mathsf{says}\ \mathsf{s} \wedge P\ \mathsf{says}\ (\mathsf{s} \rightarrow \mathsf{s}')) \rightarrow P\ \mathsf{says}\ \mathsf{s}'$

We can draw conclusions from statements made by principals.

4 *If* s *then* $P\ \mathsf{says}\ \mathsf{s}$ *for every principal* P.

True ABLP statements are supported by all principals.

Example

Given statement $\mathit{Alice}\ \mathsf{says}\ (\mathsf{s}_1 \wedge \mathsf{s}_2)$ how do we conclude that $\mathit{Alice}\ \mathsf{says}\ \mathsf{s}_1$.

Example

Given statement $\mathit{Alice}\ \mathsf{says}\ (\mathsf{s}_1 \wedge \mathsf{s}_2)$ how do we conclude that $\mathit{Alice}\ \mathsf{says}\ \mathsf{s}_1$.

We use the following steps.

$(\mathsf{s}_1 \wedge \mathsf{s}_2) \longrightarrow \mathsf{s}_1$	by (1)
$\mathit{Alice}\ \mathsf{says}\ ((\mathsf{s}_1 \wedge \mathsf{s}_2) \longrightarrow \mathsf{s}_1)$	by (4)
$\mathit{Alice}\ \mathsf{says}\ \mathsf{s}_1$	by (3)

Axioms about principals

5 $(P \wedge Q)$ says s $\equiv$ (P says s) $\wedge$ (Q says s)

6 $(P \mid Q)$ says s $\equiv$ P says (Q says s)

7 $(P = Q) \rightarrow$ (P says s $\equiv$ Q says s)

$=$ is equality on principals.

8 $(P_1 \mid (P_2 \mid P_3)) = ((P_1 \mid P_2) \mid P_3)$

Quoting is associative.

9 $(P_1 \mid (P_2 \wedge P_3)) = (P_1 \mid P_2) \wedge (P_1 \mid P_3)$

Quoting distributes over conjunction

10 $(P \Rightarrow Q) \equiv (P = P \wedge Q)$

11 $(P \text{ says } (Q \Rightarrow P)) \rightarrow (Q \Rightarrow P)$

A principal is free to choose a representative.

Example We want to conclude s from the three statements:

- $(Alice \wedge Bob)\ \mathsf{says}\ (Charlie \Rightarrow (Alice \wedge Bob))$
- $Charlie \mid Alice\ \mathsf{says}\ \mathsf{s}$
- $(Alice\ \mathsf{says}\ \mathsf{s}) \rightarrow \mathsf{s}$

$$
\begin{array}{lr}
(Alice \wedge Bob)\ \mathsf{says}\ (Charlie \Rightarrow (Alice \wedge Bob)) & \\
\quad \rightarrow (Charlie \Rightarrow (Alice \wedge Bob)) & \text{by (11)} \\
(Charlie \Rightarrow (Alice \wedge Bob)) & \text{by (2)} \\
Charlie = (Charlie \wedge Alice \wedge Bob) & \text{by (10)} \\
Charlie\ \mathsf{says}\ (Alice\ \mathsf{says}\ \mathsf{s}) & \text{by (6)} \\
(Charlie \wedge Alice \wedge Bob)\ \mathsf{says}\ (Alice\ \mathsf{says}\ \mathsf{s}) & \text{by (7,2)}
\end{array}
$$

$Alice$ says ($Alice$ says s) by (5,1,2)

$Alice$ says (($Alice$ says s)$\longrightarrow$s) by (4)

$Alice$ says s by (3)

s by (2)

Modeling Java stack inspection using ABLP

Wallach, Felten, 1998

Code can be digitally signed by a signer. We treat code, public keys and signers as principals. Stack frames created during execution of code are also treated as principals. Targets (resources to be protected) are also treated as principals.

Modeling Java stack inspection using ABLP

Wallach, Felten, 1998

Code can be digitally signed by a signer. We treat code, public keys and signers as principals. Stack frames created during execution of code are also treated as principals. Targets (resources to be protected) are also treated as principals.

If K is a public key of S then we have the statement

$$K \Rightarrow S \tag{S1}$$

Modeling Java stack inspection using ABLP

Wallach, Felten, 1998

Code can be digitally signed by a signer. We treat code, public keys and signers as principals. Stack frames created during execution of code are also treated as principals. Targets (resources to be protected) are also treated as principals.

If K is a public key of S then we have the statement

$$K \Rightarrow S \tag{S1}$$

If some code C was signed and K is the corresponding public key then we have the statement

$$K \ \mathsf{says}\ (C \Rightarrow K) \tag{S2}$$

If F is the stack frame generated for executing code C then we have the statement

$$F \Rightarrow C \tag{S3}$$

Frame credentials Φ = set of all valid statements of the form S1,S2 and S3.

If F is the stack frame generated for executing code C then we have the statement

$$F \Rightarrow C \tag{S3}$$

Frame credentials Φ = set of all valid statements of the form S1,S2 and S3.

Note that from K says $(C \Rightarrow K)$ using (11) we can conclude $C \Rightarrow K$.

Further we can show transitivity of $\Rightarrow$: given $A \Rightarrow B$ and $B \Rightarrow C$ we have:
$A = A \wedge B$ by (10)
$B = B \wedge C$ by (10)
Hence $A = A \wedge B \wedge C = A \wedge C$
Hence we have $A \Rightarrow C$

If F is the stack frame generated for executing code C then we have the statement

$$F \Rightarrow C \tag{S3}$$

Frame credentials Φ = set of all valid statements of the form S1,S2 and S3.

Note that from K says $(C \Rightarrow K)$ using (11) we can conclude $C \Rightarrow K$.

Further we can show transitivity of $\Rightarrow$: given $A \Rightarrow B$ and $B \Rightarrow C$ we have:
$A = A \wedge B$ by (10)
$B = B \wedge C$ by (10)
Hence $A = A \wedge B \wedge C = A \wedge C$
Hence we have $A \Rightarrow C$

Hence from S1, S2 and S3 we can conclude $F \Rightarrow S$.

For each target T we treat $\mathsf{Ok}(T)$ as an atomic statement.

It means that access to T is permitted.

We consider the axiom

$$(T \text{ says } \mathsf{Ok}(T)) \rightarrow \mathsf{Ok}(T) \tag{S4}$$

A target is always free to grant permission to itself.

Targets are dummy principals. They never speak, but other (non-dummy) principals representing them may speak for them.

Target credentials $\mathcal{T}$ is the set of such axioms for all targets T.

Policy for a virtual machine M is defined by a set

access credentials $\mathcal{A}_{\mathsf{M}}$ of statements of the form $P \Rightarrow T$ where P is a principal and T is a target.

This rule means that the local policy of virtual machine M allows P to access T.

Stacks

During execution, at any point of time, a stack frame F has a belief set $\mathcal{B}_F$

This is updated as follows.

Stacks

During execution, at any point of time, a stack frame F has a belief set $\mathcal{B}_F$

This is updated as follows.

Starting the program For the initial stack frame F_0

$\mathcal{B}_{F_0} = \{\mathsf{Ok}(T) \mid T \text{ is a target}\}.$

Stacks

During execution, at any point of time, a stack frame F has a belief set $\mathcal{B}_F$

This is updated as follows.

Starting the program For the initial stack frame F_0

$\mathcal{B}_{F_0} = \{\mathsf{Ok}(T) \mid T \text{ is a target}\}$.

Enabling privileges

If stack frame F calls $\mathsf{enablePrivilege}(T)$ then we update: $\mathcal{B}_F := \mathcal{B}_F \cup \{\mathsf{Ok}(T)\}$.

Stacks

During execution, at any point of time, a stack frame F has a belief set $\mathcal{B}_F$

This is updated as follows.

Starting the program For the initial stack frame F_0

$\mathcal{B}_{F_0} = \{\mathsf{Ok}(T) \mid T \text{ is a target}\}$.

Enabling privileges

If stack frame F calls $\mathsf{enablePrivilege}(T)$ then we update: $\mathcal{B}_F := \mathcal{B}_F \cup \{\mathsf{Ok}(T)\}$.

Function calls

Function call from stack frame F creates a new stack frame G.

$\mathcal{B}_G = \{F \ \mathsf{says}\ \mathsf{s} \mid \mathsf{s} \in \mathcal{B}_F\}$.

Disabling privileges

If stack frame F calls $\mathsf{disablePrivilege}(T)$ then we update

$\mathcal{B}_F := \mathcal{B}_F \setminus \{\mathsf{s} \mid \mathsf{Ok}(T) \text{ occurs in } \mathsf{s}\}$

Disabling privileges

If stack frame F calls $\mathsf{disablePrivilege}(T)$ then we update $\mathcal{B}_F := \mathcal{B}_F \setminus \{\mathsf{s} \mid \mathsf{Ok}(T) \text{ occurs in } \mathsf{s}\}$

Reverting privileges

If stack frame F calls $\mathsf{revertPrivilege}(T)$ then we update $\mathcal{B}_F := \mathcal{B}_F \setminus \{\mathsf{Ok}(T)\}$

Disabling privileges

If stack frame F calls $\mathsf{disablePrivilege}(T)$ then we update $\mathcal{B}_F := \mathcal{B}_F \setminus \{\mathsf{s} \mid \mathsf{Ok}(T) \text{ occurs in } \mathsf{s}\}$

Reverting privileges

If stack frame F calls $\mathsf{revertPrivilege}(T)$ then we update $\mathcal{B}_F := \mathcal{B}_F \setminus \{\mathsf{Ok}(T)\}$

Checking privileges

When F calls $\mathsf{checkPrivilege}(T)$ then we check that $\mathsf{Ok}(T)$ can be concluded from the set

$\Phi \cup \mathcal{T} \cup \mathcal{A}_{\mathsf{M}} \cup \{F \ \mathsf{says}\ \mathsf{s} \mid \mathsf{s} \in \mathcal{B}_F\}$.

Example Assume at the beginning that $\mathcal{B}_{F_1} = \{\}$.

Now F_1 calls $\mathsf{enablePrivilege}(T_1)$. We have $\mathcal{B}_{F_1} = \{\mathsf{Ok}(T_1)\}$.

F_1 calls $\mathsf{checkPrivilege}(T_1)$.

Hence we take the statement F_1 says $\mathsf{Ok}(T_1)$.

Let S_1 be the signer of the code which produced the frame F_1.

Then we conclude $F_1 \Rightarrow S_1$ from the frame credentials Φ.

If the access credentials set $\mathcal{A}_{\mathsf{M}}$ has a statement $S_1 \Rightarrow T_1$

then using the statement $(T_1 \text{ says } \mathsf{Ok}(T_1)) \rightarrow \mathsf{Ok}(T_1)$ from T

we conclude $\mathsf{Ok}(T_1)$.

Now F_1 makes a function call and the new frame F_2 calls $\mathsf{enablePrivilege}(T_2)$.

We have $\mathcal{B}_{F_2} = \{F_1 \text{ says } \mathsf{Ok}(T_1), \mathsf{Ok}(T_2)\}$

F_2 makes function call and the new frame F_3 calls $\mathsf{disablePrivilege}(T_1)$.

We have $\mathcal{B}_{F_3} = \{F_2 \text{ says } \mathsf{Ok}(T_2)\}$.

F_3 makes function call and the new frame F_4 calls $\mathsf{enablePrivilege}(T_2)$.

We have $\mathcal{B}_{F_4} = \{(F_3 \mid F_2) \text{ says } \mathsf{Ok}(T_2), \mathsf{Ok}(T_2)\}$.

F_4 calls $\mathsf{revertPrivilege}(T_2)$.

We have $\mathcal{B}_{F_4} = \{(F_3 \mid F_2) \text{ says } \mathsf{Ok}(T_2)\}$.

Now F_4 calls $\mathsf{checkPrivilege}T_2$.

We take the statement $(F_4 \mid F_3 \mid F_2)$ says $\mathsf{Ok}(T_2)$ i.e.

F_4 says $(F_3$ says $(F_2$ says $\mathsf{Ok}(T_2)))$.

Suppose from the frame credentials Φ imply that

$F_4 \Rightarrow S_4 \quad F_3 \Rightarrow S_3 \quad F_2 \Rightarrow S_2$

Suppose that $\mathcal{A}_{\mathsf{M}}$ further has statements

$S_4 \Rightarrow T_2 \quad S_3 \Rightarrow T_2 \quad S_2 \Rightarrow T_2$

Then we conclude:

T_2 says $(F_3$ says $(F_2$ says $\mathsf{Ok}(T_2)))$
T_2 says $(T_2$ says $(F_2$ says $\mathsf{Ok}(T_2)))$

T_2 says $(T_2$ says $(T_2$ says $\mathsf{Ok}(T_2)))$

Further $(T_2 \text{ } \mathsf{says} \text{ } \mathsf{Ok}(T_2)) \rightarrow \mathsf{Ok}(T_2)$ is in $\mathcal{T}$.

Hence T_2 says $(T_2 \text{ } \mathsf{says} \text{ } ((T_2 \text{ } \mathsf{says} \text{ } \mathsf{Ok}(T_2)) \rightarrow \mathsf{Ok}(T_2)))$.

Hence T_2 says $(T_2 \text{ } \mathsf{says} \text{ } \mathsf{Ok}(T_2))$.

Similarly T_2 says $\mathsf{Ok}(T_2)$.

Hence $\mathsf{Ok}(T_2)$.

Security protocols

For secure communication over an insecure network.

- Adversary can spy on messages,
- delete messages,
- modify messages,
- impersonate as Alice to Bob,
- deny having sent or received a message
- ...

Encrypting and decrypting messages

…the naive way:

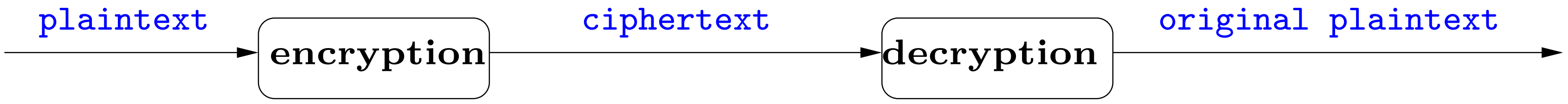
Instead of Alice ⟶ Bob:

`This is Alice.  My credit card number is 1234567890123456`

We have Alice ⟶ Bob:

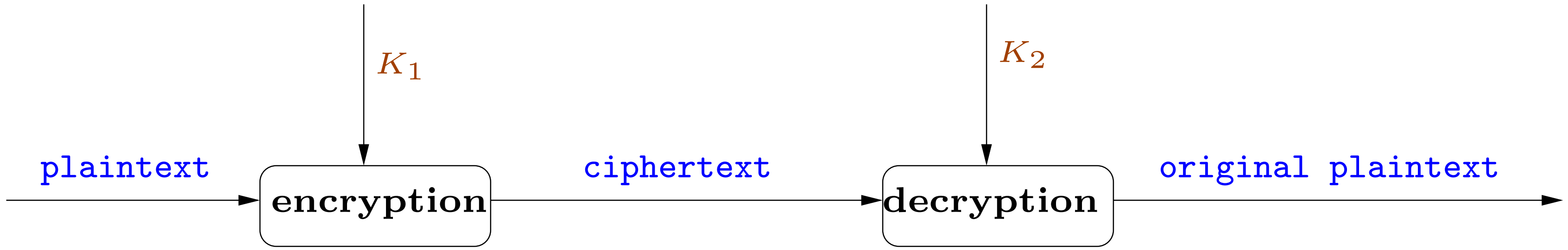
`6543210987654321 si rebmun drac tiderc yM .ecilA si sihT`

Alice and Bob agree on the method of encryption and decryption.



Cryptography with keys

Today we instead have the following picture:

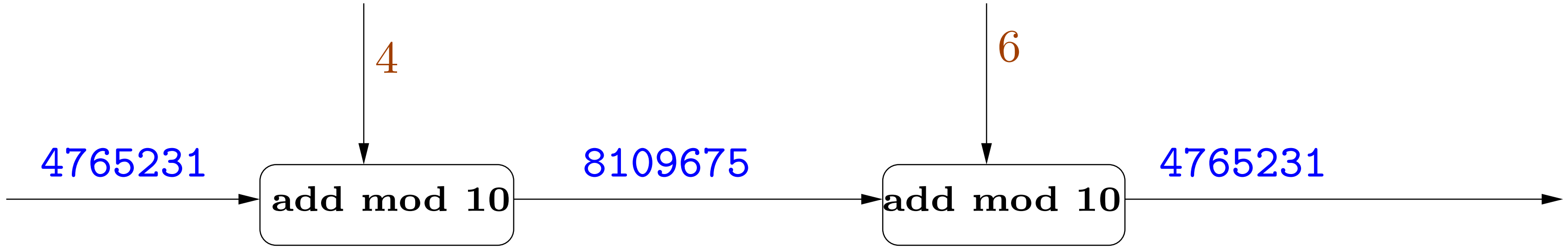


K_1 K_2

plaintext **encryption** ciphertext **decryption** original plaintext

The encryption and decryption algorithms are assumed to be publicly known.

The security lies in the (secret) keys.



4 6

4765231 **add mod 10** 8109675 **add mod 10** 4765231

Cryptography of the pre-computer age

Substitution ciphers: each character is mapped to the another character. The famous Caesar cipher: A → D, B → E, . . . , Z → C.

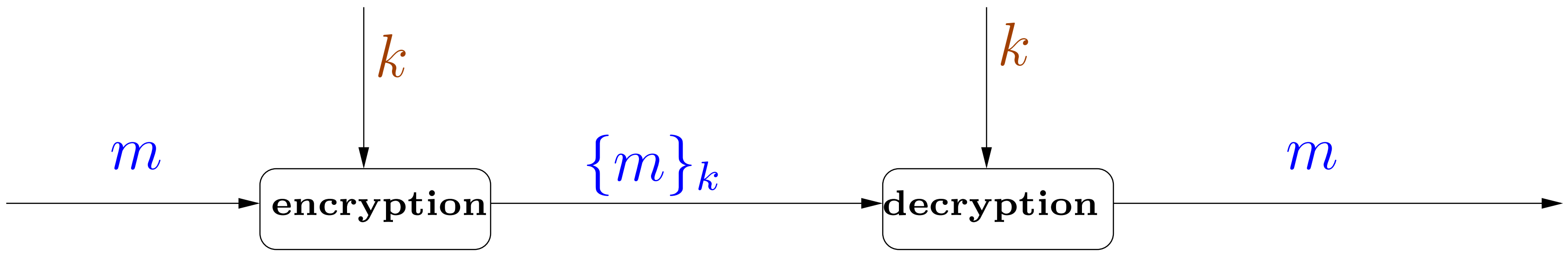
transposition cipher: shuffling around of characters.

Plaintext: `this is alice my credit card number is 1234567890123456`

```
thisisalic
emycreditc
ardnumberi
s123456789
0123456
```

Ciphertext: `teas0 hmr11 iyd22 scn33 iru44 sem55 adb66 lie7i tr8cc i9`

Private key cryptography



- The same key k is used for encryption and decryption
- Given message m and key k, we can compute the encrypted message $\{m\}_k$
- Given the encrypted message $\{m\}_k$ and the key k, we can compute the original message m

Private key cryptography

Suppose K_{ab} is a private key shared between A and B.

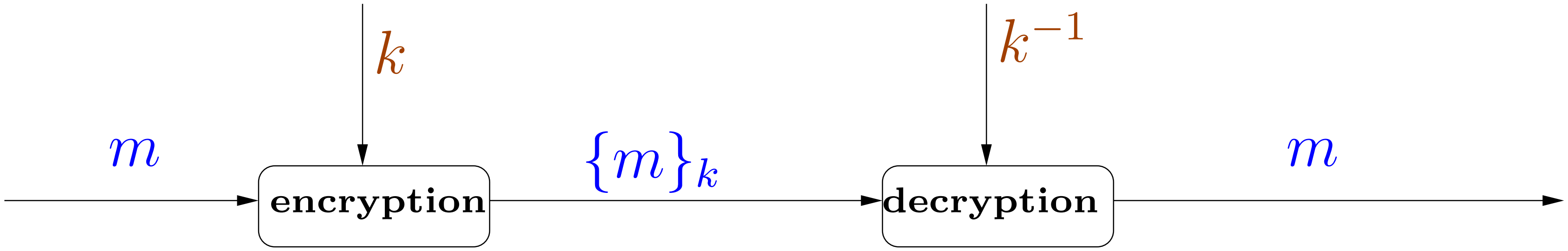
A can send a message m to B using private key cryptography:

$$A \longrightarrow B : \{m\}_{K_{ab}}$$

Only B can get back the message m.

A and B need to agree beforehand on a key K_{ab} which should not be disclosed to any one else

Public key cryptography



m → **encryption** (key k) → $\{m\}_k$ → **decryption** (key k^{-1}) → m

- A chooses pair (K_a, K_a^{-1}) of keys such that
    - messages encrypted with K_a can be decrypted with K_a^{-1}
    - K_a^{-1} cannot be calculated from K_a
- A makes K_a public: this is the public key of A
- A keeps K_a^{-1} secret: this is the private key of A

Public key cryptography

Then any B can send a message to A which only A can read:

$$B \longrightarrow A : \{m\}_{K_a}$$

Sometimes we have the additional property: messages encrypted with K_a^{-1} can be decrypted with K_a

Then A can send a message m to B

$$A \longrightarrow B : \{m\}_{K_a^{-1}}$$

and B is sure that the message m was encrypted by A. Hence we have authentication

One way hash functions

Properties of a one way hash function H:

– Given M, it is easy to compute $H(M)$ (called message digest).
– Given $H(M)$ is is difficult to find M' such that $H(M) = H(M')$.

A sends to B the message M together with the encrypted hash value $\{H(M)\}_{K_{ab}}$.

Efficient means of demonstrating authenticity, since $H(M)$ is of a fixed size.

Cryptography is not enough!

Intruder is more clever. He can attack even if the cryptographic algorithms are perfect.

Alice tells Bank to transfer £5000 to Charlie's (intruder) account:

$$A \longrightarrow B : \{A, B, \text{ transfer 5000 euros } \ldots\}_{K_{ab}}$$

- B believes that message comes from A
- Charlie has no way to decrypt the message

Cryptography is not enough!

Intruder is more clever. He can attack even if the cryptographic algorithms are perfect.

Alice tells Bank to transfer £5000 to Charlie's (intruder) account:

$$A \longrightarrow B : \{A, B,\ \text{transfer 5000 euros}\ \ldots\}_{K_{ab}}$$

- B believes that message comes from A
- Charlie has no way to decrypt the message
- But: Charlie can send the same message again to the bank

Intruder can replay known messages (freshness attack)

Solution: use session key

Generate fresh random value (nonce) for each new session and use it as a key for that session.

Solution: use session key

Generate fresh random value (nonce) for each new session and use it as a key for that session.

How to agree on a fresh key for each session?

Solution: use session key

Generate fresh random value (nonce) for each new session and use it as a key for that session.

How to agree on a fresh key for each session?

A sends to B the new key K_{ab} at the beginning of the session:

$$A \longrightarrow B : K_{ab}$$

And then uses it during that session.

Solution: use session key

Generate fresh random value (nonce) for each new session and use it as a key for that session.

How to agree on a fresh key for each session?

A sends to B the new key K_{ab} at the beginning of the session:

$$A \longrightarrow B : K_{ab}$$

And then uses it during that session.

Doesn't work. What about

$$A \longrightarrow B : \{K_{ab}\}_{K_{long}}$$

Using a long term key to agree on a session key.

A more complex solution A and B both choose a nonce each.

1. $A \longrightarrow B : \{A, N_a\}_{K_b}$
2. $B \longrightarrow A : \{N_a, N_b\}_{K_a}$
3. $A \longrightarrow B : \{N_b\}_{K_b}$

A more complex solution A and B both choose a nonce each.

1. $A \longrightarrow B : \{A, N_a\}_{K_b}$
2. $B \longrightarrow A : \{N_a, N_b\}_{K_a}$
3. $A \longrightarrow B : \{N_b\}_{K_b}$

The second message is to assure A that B is active and N_b is fresh.
The third message is to assure B that A is active and N_a is fresh.

A more complex solution A and B both choose a nonce each.

$$1. \quad A \longrightarrow B : \{A, N_a\}_{K_b}$$

$$2. \quad B \longrightarrow A : \{N_a, N_b\}_{K_a}$$

$$3. \quad A \longrightarrow B : \{N_b\}_{K_b}$$

The second message is to assure A that B is active and N_b is fresh.
The third message is to assure B that A is active and N_a is fresh.

Expected security property: N_a and N_b are known only to A and B.
Expected authentication property: A and B are assured that they are talking to each other.

$$A \longrightarrow B : \{A, B, N_a, N_b \text{ transfer 5000 euros } \ldots\}_{K_b}$$

A more complex solution A and B both choose a nonce each.

$$\begin{array}{ll} 1. & A \longrightarrow B : \{A, N_a\}_{K_b} \\ 2. & B \longrightarrow A : \{N_a, N_b\}_{K_a} \\ 3. & A \longrightarrow B : \{N_b\}_{K_b} \end{array}$$

The second message is to assure A that B is active and N_b is fresh.
The third message is to assure B that A is active and N_a is fresh.

Expected security property: N_a and N_b are known only to A and B.
Expected authentication property: A and B are assured that they are talking to each other.

$$A \longrightarrow B : \{A, B, N_a, N_b \text{ transfer 5000 euros } \ldots\}_{K_b}$$

How secure is this ? How to guarantee security ?

Cryptography and cryptographic protocols

- Cryptography deals with algorithms for encryption, decryption, random number generation, etc. Cryptographic protocols use cryptography for exchanging messages.
- Attacks against cryptographic primitives involves breaking the algorithm for encryption, etc. Attacks against cryptographic protocols may be of completely logical nature.
- Cryptographic protocols may be insecure even if the underlying cryptographic primitives are completely secure.
- Hence we often separate the study of cryptographic protocols from that of cryptographic primitives.

Difficulty in ensuring correctness of cryptographic protocols

- Infinitely many sessions
- Infinitely many participants
- Infinitely many nonces
- Sessions are interleaved
- Adversary can replace messages by any arbitrary message: infinitely branching system

Back to our example

1. $A \longrightarrow B : \{A, N_a\}_{K_b}$
2. $B \longrightarrow A : \{N_a, N_b\}_{K_a}$
3. $A \longrightarrow B : \{N_b\}_{K_b}$

Back to our example

1. $A \longrightarrow B : \{A, N_a\}_{K_b}$
2. $B \longrightarrow A : \{N_a, N_b\}_{K_a}$
3. $A \longrightarrow B : \{N_b\}_{K_b}$

This is the well-known Needham-Schroeder public-key protocol.

Published in 1978. Attack found after 17 years in 1995 by Lowe.

Man in the middle attack

$$\text{A} \xrightarrow{\{A, N_a\}_{K_c}} \text{C (A)} \xrightarrow{\{A, N_a\}_{K_b}} \text{B}$$

$$\text{A} \xleftarrow{\{N_a, N_b\}_{K_a}} \text{C (A)} \xleftarrow{\{N_a, N_b\}_{K_a}} \text{B}$$

$$\text{A} \xrightarrow{\{N_b\}_{K_c}} \text{C (A)} \xrightarrow{\{N_b\}_{K_b}} \text{B}$$

Man in the middle attack

$$\text{A} \xrightarrow{\{A, N_a\}_{K_c}} \text{C (A)} \xrightarrow{\{A, N_a\}_{K_b}} \text{B}$$

$$\text{A} \xleftarrow{\{N_a, N_b\}_{K_a}} \text{C (A)} \xleftarrow{\{N_a, N_b\}_{K_a}} \text{B}$$

$$\text{A} \xrightarrow{\{N_b\}_{K_c}} \text{C (A)} \xrightarrow{\{N_b\}_{K_b}} \text{B}$$

Even very simple protocols may have subtle flaws

Consequences

Suppose B is the server of a bank.

C, who can now pretend to be A:

$$C \longrightarrow B : \{N_a, N_b,\ \text{transfer £5000 from account of } A \text{ to account of } C\}_{K_b}$$

A fix: the Needham-Schroeder-Lowe protocol [Lowe,1985]

B includes his identity in the message he sends:

1. $A \longrightarrow B : \{A, Na\}_{K_b}$
2. $B \longrightarrow A : \{B, N_a, N_b\}_{K_a}$
3. $A \longrightarrow B : \{N_b\}_{K_b}$

A fix: the Needham-Schroeder-Lowe protocol [Lowe,1985]

B includes his identity in the message he sends:

1. $A \longrightarrow B : \{A, Na\}_{K_b}$
2. $B \longrightarrow A : \{B, N_a, N_b\}_{K_a}$
3. $A \longrightarrow B : \{N_b\}_{K_b}$

Is it secure?

A variant of the Needham-Schroeder-Lowe protocol

Suppose now we change the place of B in the second message:

1. $A \longrightarrow B : \{A, Na\}_{K_b}$
2. $B \longrightarrow A : \{N_a, N_b, B\}_{K_a}$
3. $A \longrightarrow B : \{N_b\}_{K_b}$

A variant of the Needham-Schroeder-Lowe protocol

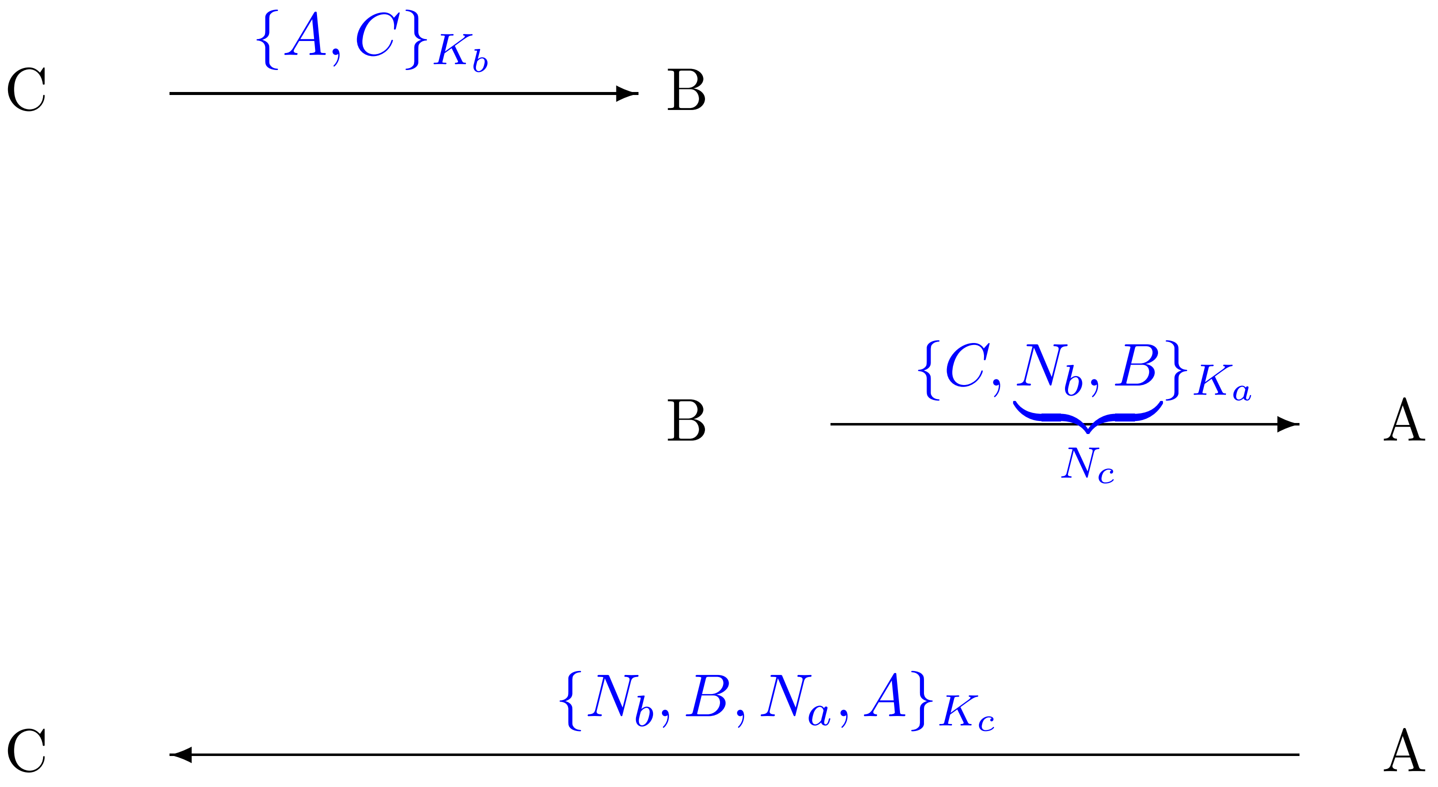
Suppose now we change the place of B in the second message:

1. $A \longrightarrow B : \{A, Na\}_{K_b}$
2. $B \longrightarrow A : \{N_a, N_b, B\}_{K_a}$
3. $A \longrightarrow B : \{N_b\}_{K_b}$

Does this affect security?

Type flaw

An attack on the variant of the Needham-Schroeder-Lowe protocol [Millen]:



C $\xrightarrow{\{A, C\}_{K_b}}$ B

B $\xrightarrow{\{C, \underbrace{N_b, B}_{N_c}\}_{K_a}}$ A

C $\xleftarrow{\{N_b, B, N_a, A\}_{K_c}}$ A

The Spi calculus

Abadi, Gordon, 1997

- Extends pi calculus which provides a language for describing processes.
- We treat protocols as processes, where messages sent and received by processes may involve encryption.
- Security is defined as equivalence between processes in the eyes of an arbitrary environment.
- Environment is also a spi calculus process.
- We study information flow to check whether secrets are leaked.

- A process may involve sequences of actions for sending and receiving messages on channels.
- A Processes may contain smaller processes running in parallel.

- A process may involve sequences of actions for sending and receiving messages on channels.
- A Processes may contain smaller processes running in parallel.

Use halt to denote a finished process: it does nothing.

We write $\mathsf{send}_c\langle M\rangle; P$ to denote a process that sends the message M on channel c after which it executes the process P.

$\mathsf{recv}_c(x); Q$ denotes a process that is listening on the channel c.

On receiving some message M on this channel then it executes process $Q[M/x]$.

The process

$$P_1 \triangleq \mathsf{recv}_c(x); \mathsf{send}_d\langle x\rangle; \mathsf{halt}$$

on receiving message M on channel c, sends M on channel d and then halts.

The process

$$P_2 \triangleq \mathsf{send}_c\langle M\rangle; \mathsf{halt}$$

sends M on channel c and halts.

The process

$$P_1 \triangleq \mathsf{recv}_c(x); \mathsf{send}_d\langle x\rangle; \mathsf{halt}$$

on receiving message M on channel c, sends M on channel d and then halts.

The process

$$P_2 \triangleq \mathsf{send}_c\langle M\rangle; \mathsf{halt}$$

sends M on channel c and halts.

Putting them in parallel gives the process

$$P_3 \triangleq P_1 \mid P_2$$

The message sent by P_1 is received by P_2. Hence P_3 as a whole can make a "silent" transition to the process $\mathsf{send}_d\langle M\rangle; \mathsf{halt}$.

Further the process

$$P_5 \triangleq P_3 \mid P_4$$

where

$$P_4 \triangleq \mathsf{recv}_d(x); \mathsf{halt}$$

can halt after making only silent transitions.

Intuitively P_5 represents the protocol

$$P_2 \longrightarrow P_1 : \quad M \quad \text{(on channel } c\text{)}$$

$$P_1 \longrightarrow P_4 : \quad M \quad \text{(on channel } d\text{)}$$

We can restrict access to channels.

The process $\mathsf{new}\ c; P$ creates a fresh channel c and can be used inside process P. No outside process can access c.

(c is like a bound variable whose scope is inside P)

We consider processes to be the same after renaming of bound names.

Consider the process

$$(\mathsf{new}\ c; \mathsf{send}_c\langle M\rangle; \mathsf{halt}) \mid (\mathsf{recv}_c(x); \mathsf{halt})$$

No communication happens between the two smaller processes.

The above process is the same as the following one.

$$(\mathsf{new}\ d; \mathsf{send}_d\langle M\rangle; \mathsf{halt}) \mid (\mathsf{recv}_c(x); \mathsf{halt})$$

Hence new allows us to create channels for secure communication.

Consider the process

$$\mathsf{new}\ c; (\mathsf{send}_c\langle M\rangle; \mathsf{halt} \mid \mathsf{recv}_c(x); P \mid \mathsf{recv}_c(x); Q)$$

Communication can take place between first and second subprocess to create the process $\mathsf{new}\ c; (P[M/x] \mid \mathsf{recv}_c(x); Q)$

Or communication can take place between first and third subprocess to create the process $\mathsf{new}\ c; (\mathsf{recv}_c(x); P \mid Q[M/x])$

Hence new allows us to create channels for secure communication.

Consider the process

$$\mathsf{new}\ c; (\mathsf{send}_c\langle M\rangle; \mathsf{halt} \mid \mathsf{recv}_c(x); P \mid \mathsf{recv}_c(x); Q)$$

Communication can take place between first and second subprocess to create the process $\mathsf{new}\ c; (P[M/x] \mid \mathsf{recv}_c(x); Q)$

Or communication can take place between first and third subprocess to create the process $\mathsf{new}\ c; (\mathsf{recv}_c(x); P \mid Q[M/x])$

However the process

$$(\mathsf{new}\ c; (\mathsf{send}_c\langle M\rangle; \mathsf{halt} \mid \mathsf{recv}_c(x); P)) \mid \mathsf{recv}_c(x); Q$$

can only lead to the process $(\mathsf{new}\ c; P[M/x]) \mid \mathsf{recv}_c(x); Q$

Channels can also be sent as messages. Consider the following protocol where c_{AB} is a freshly created channel whereas c_{AS} and c_{SB} are long term channels.

$$A \longrightarrow S : c_{AB} \text{ on } c_{AS}$$

$$S \longrightarrow B : c_{AB} \text{ on } c_{SB}$$

$$A \longrightarrow B : M \text{ on } c_{AB}$$

can be represented as follows where $F(y)$ is a process involving variable y.

$$\begin{aligned}
A &\triangleq \mathsf{new}\ c_{AB}; \mathsf{send}_{c_{AS}}\langle c_{AB}\rangle; \mathsf{send}_{c_{AB}}\langle M\rangle.\mathsf{halt} \\
S &\triangleq \mathsf{recv}_{c_{AS}}(x); \mathsf{send}_{c_{SB}}\langle x\rangle; \mathsf{halt} \\
B &\triangleq \mathsf{recv}_{c_{SB}}(x); \mathsf{recv}_{x}(y); F(y) \\
P &\triangleq \mathsf{new}\ c_{AS}; \mathsf{new}\ c_{SB}; (A \mid S \mid B)
\end{aligned}$$

P makes silent transitions to $\mathsf{new}\ c_{AS}; \mathsf{new}\ c_{SB}; F(M)$.

Processes can perform computations like

- encryption, decryption (we will deal with only symmetric key encryption)
- pairing, unpairing
- increments, decrements
- checking equality of messages

Processes can perform computations like

- encryption, decryption (we will deal with only symmetric key encryption)
- pairing, unpairing
- increments, decrements
- checking equality of messages

The process

$\mathsf{recv}_c(x_1, x_2, x_3); \mathsf{case}\ x_1\ \mathsf{of}$
$\quad \{y_1\}_K : \mathsf{check}\ (y_1 == x_2); \mathsf{send}_c\langle y_1, \mathsf{succ}\ (x_3)\rangle; \mathsf{halt}$

receives an input of the form $\{M\}_K, M, N$ on channel c and sends out $y_1, \mathsf{succ}\ (x_3)$ on channel c.

The syntax

$M ::=$	term
n	name
(M, N)	pair
0	zero
$\mathsf{succ}\ (M)$	successor
$\{M_1, \ldots, M_k\}_N$	encryption
x	variable

$P ::=$	process
$\mathsf{send}_M\langle N_1, \ldots, N_k\rangle; P$	output
$\mathsf{recv}_M(x_1, \ldots, x_k); P$	input
halt	halt
$P \mid Q$	parallel composition
$\mathsf{repeat}\ P$	replication
$\mathsf{new}\ n; P$	restriction
$\mathsf{check}\ (M == N); P$	comparison
$\mathsf{let}\ (x, y) = M; P$	unpairing
$\mathsf{case}\ M\ \mathsf{of}\ 0 : P,\ \mathsf{succ}\ (x) : Q$	integer case analysis
$\mathsf{case}\ M\ \mathsf{of}\ \{x_1, \ldots, x_k\}_N : P$	decryption

Intuitively, $\mathsf{repeat}\ P$ represents infinitely many copies of P running in parallel.

In other words we can consider $\mathsf{repeat}\ P$ to represent $P \mid P \mid P \mid \ldots$

Consider

$$\begin{aligned} P &\triangleq \mathsf{recv}_c(x); \mathsf{halt} \\ P_1 &\triangleq \mathsf{send}_c(M_1); \mathsf{halt} \\ P_2 &\triangleq \mathsf{send}_c(M_2); \mathsf{halt} \end{aligned}$$

The process

$$P_1 \mid P_2 \mid \mathsf{repeat}\ P$$

can make silent transitions (internal communication) to create the process

$$\mathsf{repeat}\ P$$

A one message protocol using cryptography, where K_{AB} is a symmetric key shared between A and B for private communication.

$$A \longrightarrow B : \{M\}_{K_{AB}} \text{ on } c_{AB}$$

This can be represented as

$$\begin{aligned} A &\triangleq \mathsf{send}_{c_{AB}}\langle\{M\}_{K_{AB}}\rangle; \mathsf{halt} \\ B &\triangleq \mathsf{recv}_{c_{AB}}(x); \mathsf{case}\ x\ \mathsf{of}\ \{y\}_{K_{AB}} : F(y) \\ P &\triangleq \mathsf{new}\ K_{AB}; (A \mid B) \end{aligned}$$

The key K_{AB} is restricted, only A and B can use it.

The channel c_{AB} is public. Other principals may send messages on it or listen on it.

P can make silent transitions to $\mathsf{new}\ K_{AB}; F(M)$.

Formal semantics

We now need to define how processes execute.

For example we would like

$$\mathsf{send}_c\langle M\rangle; P \mid \mathsf{recv}_c(x); Q \xrightarrow{\tau} P \mid Q[M/x]$$

where τ denotes a silent action (internal communication).

Let $fn(M)$ and $fn(P)$ be the set of free names in term M and process P respectively.

Let $fv(M)$ and $fv(P)$ be the set of free variables in term M and process P respectively.

Closed processes are processes without any free variables.

Let $P \triangleq \mathsf{new}\ c; \mathsf{new}\ K; \mathsf{recv}_d(x); \mathsf{case}\ x\ \mathsf{of}\ \{y\}_{K'} : \mathsf{send}_d\langle\{y\}_K, z, c\rangle; \mathsf{halt}$.

We have

$$fn(\mathsf{send}_d\langle\{y\}_K, z, c\rangle; \mathsf{halt}) = \{c, d, K\}$$

$$fv(\mathsf{send}_d\langle\{y\}_K, z, c\rangle; \mathsf{halt}) = \{y, z\}$$

$$fn(\mathsf{case}\ x\ \mathsf{of}\ \{y\}_{K'} : \mathsf{send}_d\langle\{y\}_K, z, c\rangle; \mathsf{halt}) = \{c, d, K, K'\}$$

$$fv(\mathsf{case}\ x\ \mathsf{of}\ \{y\}_{K'} : \mathsf{send}_d\langle\{y\}_K, z, c\rangle; \mathsf{halt}) = \{x, z\}$$

$$fn(P) = \{d, K'\}$$

$$fv(P) = \{z\}$$

$$fn(\{y\}_K) = \{K\}$$

$$fv(\{y\}_K) = \{y\}$$

First we define reduction relation $>$ on closed processes:

$$\begin{array}{rll}
\mathsf{repeat}\ P & > P \mid \mathsf{repeat}\ P & \text{(R-Repeat)} \\
\mathsf{check}\ (M == M); P & > P & \text{(R-Check)} \\
\mathsf{let}\ (x, y) = (M, N); P & > P[M/x, N/y] & \text{(R-Let)} \\
\mathsf{case}\ 0\ \mathsf{of}\ 0 : P,\ \mathsf{succ}\ (x) : Q & > P & \text{(R-Zero)} \\
\mathsf{case}\ \mathsf{succ}\ (M)\ \mathsf{of}\ 0 : P,\ \mathsf{succ}\ (x) : Q & > Q[M/x] & \text{(R-Succ)} \\
\mathsf{case}\ \{M\}_N\ \mathsf{of}\ \{x\}_N : P & > P[M/x] & \text{(R-decrypt)}
\end{array}$$

When these rules cannot be applied, it means that the process cannot be simplified.

The following processes cannot be simplified, hence cannot be executed further.

$\mathsf{check}\ (0 == \mathsf{succ}\ (0); P$ (comparison fails).

$\mathsf{let}\ (x, y) = 0; P$ (unpairing fails)

$\mathsf{case}\ (M, N)\ \mathsf{of}\ 0 : P,\ \mathsf{succ}\ (x) : Q$ (not an integer)

$\mathsf{case}\ (M, N)\ \mathsf{of}\ \{x, y\}_K : P$ (not an encrypted message)

$\mathsf{case}\ \{M, N\}_{K'}\ \mathsf{of}\ \{x, y\}_K : P$ where $K \neq K'$

When these rules cannot be applied, it means that the process cannot be simplified.

The following processes cannot be simplified, hence cannot be executed further.

$\mathsf{check}\ (0 == \mathsf{succ}\ (0); P$ (comparison fails).

$\mathsf{let}\ (x, y) = 0; P$ (unpairing fails)

$\mathsf{case}\ (M, N)\ \mathsf{of}\ 0 : P,\ \mathsf{succ}\ (x) : Q$ (not an integer)

$\mathsf{case}\ (M, N)\ \mathsf{of}\ \{x, y\}_K : P$ (not an encrypted message)

$\mathsf{case}\ \{M, N\}_{K'}\ \mathsf{of}\ \{x, y\}_K : P$ where $K \neq K'$

This is also based on the perfect cryptography assumption: distinct terms represent distinct messages.

A barb β is either

- a name n (representing input on channel n), or
- a co-name $\overline{n}$ (representing output on channel n)

An action is either

- a barb (representing input or output to the outside world), or
- τ (representing a silent action i.e. internal communication)

We write $P \xrightarrow{\alpha} Q$ to mean that P makes action α after which Q is the remaining process that is left to be executed.

Commitment relation Consider again $\mathsf{send}_c\langle M\rangle; P \mid \mathsf{recv}_c(x); Q$

Commitment relation Consider again $\mathsf{send}_c\langle M\rangle; P \mid \mathsf{recv}_c(x); Q$

The first subprocess makes an output action on channel c.

We will represent it as $\mathsf{send}_c\langle M\rangle; P \xrightarrow{\overline{c}} \langle M\rangle P$.

$\langle M\rangle P$ is called a concretion: it represents a commitment to output message M after which P will be executed.

Commitment relation Consider again $\mathsf{send}_c\langle M\rangle; P \mid \mathsf{recv}_c(x); Q$

The first subprocess makes an output action on channel c.

We will represent it as $\mathsf{send}_c\langle M\rangle; P \xrightarrow{\overline{c}} \langle M\rangle P$.

$\langle M\rangle P$ is called a concretion: it represents a commitment to output message M after which P will be executed.

The second subprocess makes an input action on channel c.

We will represent it as $\mathsf{recv}_c(x); Q \xrightarrow{c} (x)Q$.

$(x)Q$ is called an abstraction:it represents a commitment to input some x after which P will be executed.

Commitment relation Consider again $\mathsf{send}_c\langle M\rangle; P \mid \mathsf{recv}_c(x); Q$

The first subprocess makes an output action on channel c.

We will represent it as $\mathsf{send}_c\langle M\rangle; P \xrightarrow{\overline{c}} \langle M\rangle P$.

$\langle M\rangle P$ is called a concretion: it represents a commitment to output message M after which P will be executed.

The second subprocess makes an input action on channel c.

We will represent it as $\mathsf{recv}_c(x); Q \xrightarrow{c} (x)Q$.

$(x)Q$ is called an abstraction:it represents a commitment to input some x after which P will be executed.

Abstractions and concretions can be combined:

$\langle M\rangle P \mathbin{@} (x)Q = P \mid Q[M/x]$

Formally an abstraction F is of the form

$$(x_1, \ldots, x_k)P$$

where $k \geq 0$ and P is a process.

A concretion C is of the form

$$(\mathsf{new}\ n_1, \ldots, n_l)\langle M_1, \ldots, M_k\rangle P$$

where $n_1, \ldots, n_l$ are names, $l, k \geq 0$ and P is a process.

Formally an abstraction F is of the form

$$(x_1, \ldots, x_k)P$$

where $k \geq 0$ and P is a process.

A concretion C is of the form

$$(\mathsf{new}\ n_1, \ldots, n_l)\langle M_1, \ldots, M_k \rangle P$$

where $n_1, \ldots, n_l$ are names, $l, k \geq 0$ and P is a process.

For $F \triangleq (x_1, \ldots, x_k)P$ and $C \triangleq (\mathsf{new}\ n_1, \ldots, n_l)\langle M_1, \ldots, M_k \rangle Q$ with $\{n_1, \ldots, n_l\} \cap fn(P) = \emptyset$ we define interaction of F and C as

$$\boxed{\begin{array}{l} F\ @\ C \triangleq \mathsf{new}\ n_1; \ldots \mathsf{new}\ n_l; (P[M_1/x_1, \ldots, M_k/x_k] \mid Q) \\ C\ @\ F \triangleq \mathsf{new}\ n_1; \ldots \mathsf{new}\ n_l; (Q \mid P[M_1/x_1, \ldots, M_k/x_k]) \end{array}}$$

An agent A is an abstraction, concretion or a process.
We write the commitment relation as $P \xrightarrow{\alpha} A$ where P is a closed process, A is a closed agent $(fv(A) = \emptyset)$ and α is an action.

An agent A is an abstraction, concretion or a process.

We write the commitment relation as $P \xrightarrow{\alpha} A$ where P is a closed process, A is a closed agent ($fv(A) = \emptyset$) and α is an action.

$$\mathsf{send}_m\langle M_1, \ldots, M_k\rangle; P \xrightarrow{\overline{m}} (\mathsf{new}\ )\langle M_1, \ldots, M_k\rangle P \qquad \text{(C-Out)}$$

An agent A is an abstraction, concretion or a process.

We write the commitment relation as $P \xrightarrow{\alpha} A$ where P is a closed process, A is a closed agent ($fv(A) = \emptyset$) and α is an action.

$$\mathsf{send}_m\langle M_1, \ldots, M_k\rangle; P \xrightarrow{\overline{m}} (\mathsf{new}\ )\langle M_1, \ldots, M_k\rangle P \qquad \text{(C-Out)}$$

$$\mathsf{recv}_m(x_1, \ldots, x_k); P \xrightarrow{m} (x_1, \ldots, x_k)P \qquad \text{(C-In)}$$

An agent A is an abstraction, concretion or a process.

We write the commitment relation as $P \xrightarrow{\alpha} A$ where P is a closed process, A is a closed agent ($fv(A) = \emptyset$) and α is an action.

$$\mathsf{send}_m\langle M_1, \ldots, M_k\rangle; P \xrightarrow{\overline{m}} (\mathsf{new}\ )\langle M_1, \ldots, M_k\rangle P \qquad \text{(C-Out)}$$

$$\mathsf{recv}_m(x_1, \ldots, x_k); P \xrightarrow{m} (x_1, \ldots, x_k)P \qquad \text{(C-In)}$$

$$\frac{P \xrightarrow{m} F \quad Q \xrightarrow{\overline{m}} C}{P \mid Q \xrightarrow{\tau} F \mathbin{@} C}\ \text{(C-Inter1)}$$

An agent A is an abstraction, concretion or a process.

We write the commitment relation as $P \xrightarrow{\alpha} A$ where P is a closed process, A is a closed agent ($fv(A) = \emptyset$) and α is an action.

$$\mathsf{send}_m\langle M_1, \ldots, M_k\rangle; P \xrightarrow{\overline{m}} (\mathsf{new}\ )\langle M_1, \ldots, M_k\rangle P \qquad \text{(C-Out)}$$

$$\mathsf{recv}_m(x_1, \ldots, x_k); P \xrightarrow{m} (x_1, \ldots, x_k)P \qquad \text{(C-In)}$$

$$\frac{P \xrightarrow{m} F \quad Q \xrightarrow{\overline{m}} C}{P \mid Q \xrightarrow{\tau} F \mathbin{@} C} \text{(C-Inter1)}$$

$$\frac{P \xrightarrow{\overline{m}} C \quad Q \xrightarrow{m} F}{P \mid Q \xrightarrow{\tau} C \mathbin{@} F} \text{(C-Inter2)}$$

Example

Define

$P \triangleq \mathsf{send}_c\langle \mathsf{succ}\ (0)\rangle; \mathsf{halt}$

$Q \triangleq \mathsf{recv}_c(x); \mathsf{case}\ x\ \mathsf{of}\ 0 : \mathsf{halt},\ \mathsf{succ}\ (y) : (\mathsf{send}_d\langle y\rangle; \mathsf{halt})$

From our rules we have

Example

Define

$P \triangleq \mathsf{send}_c\langle \mathsf{succ}\ (0)\rangle; \mathsf{halt}$

$Q \triangleq \mathsf{recv}_c(x); \mathsf{case}\ x\ \mathsf{of}\ 0 : \mathsf{halt},\ \mathsf{succ}\ (y) : (\mathsf{send}_d\langle y\rangle; \mathsf{halt})$

From our rules we have

$$P \quad \xrightarrow{\bar{c}} \langle \mathsf{succ}\ (0)\rangle \mathsf{halt}$$

$$(\langle M_1, \ldots, M_k\rangle P' \text{ denotes } (\mathsf{new}\ )\langle M_1, \ldots, M_k\rangle P')$$

Example

Define

$P \triangleq \mathsf{send}_c\langle \mathsf{succ}\ (0)\rangle; \mathsf{halt}$

$Q \triangleq \mathsf{recv}_c(x); \mathsf{case}\ x\ \mathsf{of}\ 0 : \mathsf{halt},\ \mathsf{succ}\ (y) : (\mathsf{send}_d\langle y\rangle; \mathsf{halt})$

From our rules we have

$$P \quad \xrightarrow{\bar{c}} \langle \mathsf{succ}\ (0)\rangle \mathsf{halt}$$

$$(\langle M_1, \ldots, M_k\rangle P' \text{ denotes } (\mathsf{new}\ )\langle M_1, \ldots, M_k\rangle P')$$

$$Q \quad \xrightarrow{c} (x)\mathsf{case}\ x\ \mathsf{of}\ 0 : \mathsf{halt},\ \mathsf{succ}\ (y) : (\mathsf{send}_d\langle y\rangle; \mathsf{halt})$$

Example

Define

$P \triangleq \mathsf{send}_c\langle\mathsf{succ}\ (0)\rangle; \mathsf{halt}$

$Q \triangleq \mathsf{recv}_c(x); \mathsf{case}\ x\ \mathsf{of}\ 0 : \mathsf{halt},\ \mathsf{succ}\ (y) : (\mathsf{send}_d\langle y\rangle; \mathsf{halt})$

From our rules we have

$$P \xrightarrow{\bar{c}} \langle\mathsf{succ}\ (0)\rangle\mathsf{halt}$$

$$(\langle M_1, \ldots, M_k\rangle P' \text{ denotes } (\mathsf{new}\ )\langle M_1, \ldots, M_k\rangle P')$$

$$Q \xrightarrow{c} (x)\mathsf{case}\ x\ \mathsf{of}\ 0 : \mathsf{halt},\ \mathsf{succ}\ (y) : (\mathsf{send}_d\langle y\rangle; \mathsf{halt})$$

$$P \mid Q \xrightarrow{\tau} \mathsf{halt} \mid \mathsf{case}\ \mathsf{succ}\ (0)\ \mathsf{of}\ 0 : \mathsf{halt},\ \mathsf{succ}\ (y) : (\mathsf{send}_d\langle y\rangle; \mathsf{halt})$$

Example

Define

$P \triangleq \mathsf{send}_c\langle \mathsf{succ}\ (0)\rangle; \mathsf{halt}$

$Q \triangleq \mathsf{recv}_c(x); \mathsf{case}\ x\ \mathsf{of}\ 0 : \mathsf{halt},\ \mathsf{succ}\ (y) : (\mathsf{send}_d\langle y\rangle; \mathsf{halt})$

From our rules we have

$$P \xrightarrow{\overline{c}} \langle \mathsf{succ}\ (0)\rangle \mathsf{halt}$$

$(\langle M_1, \ldots, M_k\rangle P'$ denotes $(\mathsf{new}\ )\langle M_1, \ldots, M_k\rangle P')$

$$Q \xrightarrow{c} (x)\mathsf{case}\ x\ \mathsf{of}\ 0 : \mathsf{halt},\ \mathsf{succ}\ (y) : (\mathsf{send}_d\langle y\rangle; \mathsf{halt})$$

$$P \mid Q \xrightarrow{\tau} \mathsf{halt} \mid \mathsf{case}\ \mathsf{succ}\ (0)\ \mathsf{of}\ 0 : \mathsf{halt},\ \mathsf{succ}\ (y) : (\mathsf{send}_d\langle y\rangle; \mathsf{halt})$$

$$\xrightarrow{\overline{d}} \langle 0\rangle(\mathsf{halt} \mid \mathsf{halt})$$ using the following rules…

$$\frac{P > Q \quad Q \xrightarrow{\alpha} A}{P \xrightarrow{\alpha} A}\ \text{(C-Red)}$$

$$\frac{P \xrightarrow{\alpha} A}{P \mid Q \xrightarrow{\alpha} A \mid Q}\ \text{(C-Par1)} \qquad \frac{Q \xrightarrow{\alpha} A}{P \mid Q \xrightarrow{\alpha} P \mid A}\ \text{(C-Par2)}$$

where

$$P_1 \mid (x_1, \ldots, x_k)P_2 \triangleq (x_1, \ldots, x_k)(P_1 \mid P_2)$$

$$P_1 \mid (\mathsf{new}\ n_1, \ldots, n_k)\langle M_1, \ldots, M_l\rangle P_2 \triangleq (\mathsf{new}\ n_1, \ldots, n_k)\langle M_1, \ldots, M_l\rangle(P_1 \mid P_2)$$

provided that $x_1, \ldots, x_k \notin fv(P_1)$ and $n_1, \ldots, n_k \notin fn(P_1)$

For the previous example we have using (R-Succ):

$$\mathsf{case}\ \mathsf{succ}\ (0)\ \mathsf{of}\ 0 : \mathsf{halt},\ \mathsf{succ}\ (y) : (\mathsf{send}_d\langle y\rangle; \mathsf{halt}) > \mathsf{send}_d\langle 0\rangle; \mathsf{halt}$$

and using (C-Out):

$$\mathsf{send}_d\langle 0\rangle; \mathsf{halt} \xrightarrow{\bar{d}} \langle 0\rangle\mathsf{halt}$$

hence using (C-Red):

$$\mathsf{case}\ \mathsf{succ}\ (0)\ \mathsf{of}\ 0 : \mathsf{halt},\ \mathsf{succ}\ (y) : (\mathsf{send}_d\langle y\rangle; \mathsf{halt}) \xrightarrow{\bar{d}} \langle 0\rangle\mathsf{halt}$$

hence using (C-Par2):

$$\mathsf{halt} \mid \mathsf{case}\ \mathsf{succ}\ (0)\ \mathsf{of}\ 0 : \mathsf{halt},\ \mathsf{succ}\ (y) : (\mathsf{send}_d\langle y\rangle; \mathsf{halt}) \xrightarrow{\bar{d}} \mathsf{halt} \mid \langle 0\rangle\mathsf{halt}$$
$$= \langle 0\rangle(\mathsf{halt} \mid \mathsf{halt})$$

Consider $P \triangleq (\mathsf{recv}_c(x); P_1) \mid \mathsf{new}\ c; (\mathsf{send}_c\langle 0\rangle; P_2 \mid \mathsf{recv}_c(x); P_3)$

We would like $P \xrightarrow{\tau} (\mathsf{recv}_c(x); P_1) \mid \mathsf{new}\ c; (P_2 \mid P_3[0/x])$

but not $P \xrightarrow{\tau} P_1[0/x] \mid \mathsf{new}\ n; (P_2 \mid \mathsf{recv}_c(x); P_3)$

Consider $P \triangleq (\mathsf{recv}_c(x); P_1) \mid \mathsf{new}\ c; (\mathsf{send}_c\langle 0\rangle; P_2 \mid \mathsf{recv}_c(x); P_3)$

We would like $P \xrightarrow{\tau} (\mathsf{recv}_c(x); P_1) \mid \mathsf{new}\ c; (P_2 \mid P_3[0/x])$

but not $P \xrightarrow{\tau} P_1[0/x] \mid \mathsf{new}\ n; (P_2 \mid \mathsf{recv}_c(x); P_3)$

Hence we have the rule

$$\boxed{\dfrac{P \xrightarrow{\alpha} A \quad \alpha \notin \{n, \overline{n}\}}{\mathsf{new}\ n; P \xrightarrow{\alpha} \mathsf{new}\ n; A}\ (\text{C-New})}$$

where

$$(\mathsf{new}\ m)(x_1, \ldots, x_k)P \triangleq (x_1, \ldots, x_k)\mathsf{new}\ m; P$$

$$(\mathsf{new}\ m)(\mathsf{new}\ m_1, \ldots, m_k)\langle M_1, \ldots, M_l\rangle P \triangleq (\mathsf{new}\ m, m_1, \ldots, m_k)\langle M_1, \ldots, M_l\rangle P$$

provided that $m \notin \{m_1, \ldots, m_k\}$

We have $\mathsf{send}_c\langle 0\rangle; P_2 \xrightarrow{\overline{c}} \langle 0\rangle P_2$

and $\mathsf{recv}_c(x); P_3 \xrightarrow{c} (x)P_3$

hence $\mathsf{send}_c\langle 0\rangle; P_2 \mid \mathsf{recv}_c(x); P_3 \xrightarrow{\tau} \langle 0\rangle P_2 \mathbin{@} (x)P_3 = P_2 \mid P_3[0/x]$

Since $\tau \notin \{\overline{c}, c\}$

hence $\mathsf{new}\ c; (\mathsf{send}_c\langle 0\rangle; P_2 \mid \mathsf{recv}_c(x); P_3) \xrightarrow{\tau} \mathsf{new}\ c; (P_2 \mid P_3[0/x])$

Hence $(\mathsf{recv}_c(x); P_1) \mid \mathsf{new}\ c; (\mathsf{send}_c\langle 0\rangle; P_2 \mid \mathsf{recv}_c(x); P_3)$

$\xrightarrow{\tau} (\mathsf{recv}_c(x); P_1) \mid \mathsf{new}\ c; (P_2 \mid P_3[0/x])$

Consider $P \triangleq (\mathsf{new}\ K; \mathsf{send}_c\langle K\rangle; \mathsf{halt}) \mid (\mathsf{recv}_c(x); \mathsf{send}_d\langle x\rangle; \mathsf{halt})$

We have $\mathsf{send}_c\langle K\rangle; \mathsf{halt} \xrightarrow{\overline{c}} (\mathsf{new}\ )\langle K\rangle \mathsf{halt}$

hence $\mathsf{new}\ K; \mathsf{send}_c\langle K\rangle; \mathsf{halt} \xrightarrow{\overline{c}} \mathsf{new}\ K; (\mathsf{new}\ )\langle K\rangle \mathsf{halt} = (\mathsf{new}\ K)\langle K\rangle \mathsf{halt}$

Also $\mathsf{recv}_c(x); \mathsf{send}_d\langle x\rangle; \mathsf{halt} \xrightarrow{c} (x)\mathsf{send}_d\langle x\rangle; \mathsf{halt}$

Hence
$P \xrightarrow{\tau} (\mathsf{new}\ K)\langle K\rangle \mathsf{halt}\ @\ (x)\mathsf{send}_d\langle x\rangle; \mathsf{halt} = (\mathsf{new}\ K)(\mathsf{halt} \mid \mathsf{send}_d\langle K\rangle; \mathsf{halt})$

Equivalence on processes

A test is of the form (Q, β) where Q is a closed process and β is a barb.

A process P passes the test (Q, β) iff

$$(P \mid Q) \xrightarrow{\tau} Q_1 \ldots \xrightarrow{\tau} Q_n \xrightarrow{\beta} A$$

for some $n \geq 0$, some processes $Q_1, \ldots, Q_n$ and some agent A.

Q is the "environment" and we test whether the process together with the environment inputs or outputs on a particular channel.

Testing preorder $P_1 \sqsubseteq P_2$ iff for every test (Q, β), if P_1 passes (Q, β) then P_2 passes (Q, β).

Testing equivalence $P_1 \simeq P_2$ iff $P_1 \sqsubseteq P_2$ and $P_2 \sqsubseteq P_1$.

Secrecy

Consider process P with only free variable x.

We will consider x as secret if for all terms M, M' we have $P[M/x] \simeq P[M'/x]$.

I.e. an observer cannot detect any changes in the value of x.

Example Consider $P \triangleq \mathsf{send}_c\langle x\rangle; \mathsf{halt}$.

x is being sent out on a public channel. Consider test $(Q, \overline{d})$ where

environment $Q \triangleq \mathsf{recv}_c(x); \mathsf{check}\ (x == 0); \mathsf{send}_d\langle \mathsf{halt}\rangle; \mathsf{halt}$.

We have $P[0/x] \mid Q \xrightarrow{\tau} \mathsf{halt} \mid \mathsf{send}_d\langle 0\rangle; \mathsf{halt} \xrightarrow{\overline{d}} \langle 0\rangle(\mathsf{halt} \mid \mathsf{halt})$.

Hence $P[0/x]$ passes the test. However $P[\mathsf{succ}\ (0)/x]$ fails the test.

Hence P does not preserve secrecy of x.

Information flow analysis for the Spi-calculus

We classify data into three classes

secret	data which should not be leaked
public	data which can be communicated to anyone
any	arbitrary data

Subsumption relation on classes:

$$\begin{aligned} \mathsf{secret} &\preceq \mathsf{any} \\ \mathsf{public} &\preceq \mathsf{any} \\ T &\preceq T \qquad \text{for } T \in \{\mathsf{secret}, \mathsf{public}, \mathsf{any}\} \end{aligned}$$

An environment E provides information about the classes to which names and variables belong.

We define typing rules for the following kinds of judgments

	$\vdash E$	environment E is well formed
E	$\vdash M : T$	term M is of class T in environment E
E	$\vdash P$	process P is well typed in environment E

E.g. secret data should not be sent on public channels.

Data of level any should be protected as if it is of level secret, but can be exploited only as of it had level public.

Our goal is to define typing rules to filter out processes that leak secrets.

Informally we would like to show that if environment E has only any variables and public names and $E \vdash P$ then P does not leak any variables $x \in dom(E)$.

Our previous example:

$P \triangleq \mathsf{send}_c\langle x\rangle; \mathsf{halt}$

Consider $E = \{x : \mathsf{any}, c : \mathsf{public} :: L_1, d : \mathsf{public} :: L_2\}$

(L_1 and L_2 will be explained later.)

x is of level any but is sent out on c of level public, which will be forbidden by our typing rules.

Consider protocol

$A \longrightarrow S : A, B$

$S \longrightarrow A : \{A, B, Na, \{Nb\}_{K_{sb}}\}_{K_{sa}}$

$A \longrightarrow B : \{Nb\}_{K_s b}$

A principal X may play the role of A in one session and of B in another session.

We need a clear way of distinguishing the messages received and their components.

This is important only for messages sent on secret channels and for messages encrypted with public keys.

We adopt the following standard format:

messages sent on secret channels should have three components of levels secret, any and public respectively.

Consider protocol

$B \longrightarrow A : Nb$

$A \longrightarrow B : \{M, Nb\}_{K_{ab}}$

By replaying nonces, an attacker can find out whether the same M is sent more than once, or different ones. Hence he gets

some partial information about the contents of the messages.

To prevent this we include an extra fresh nonce (confounder) in each message encrypted with secret keys.

$A \longrightarrow B : \{M, Nb, Na\}_{K_{ab}}$

We adopt the following standard format for messages encrypted with secret keys: $\{M_1, M_2, M_3, n\}_K$

where M_1 has level secret, M_2 has level any, M_3 has level public,

and n is the confounder.

n can be used as confounder only in this term and nowhere else.

This information is remembered by the environment E.

I.e. if $n : T :: \{M_1, M_2, M_3, n\}_K \in E$ then

we know that n is used as a confounder only in that message.

The typing rules

The empty environment is denoted $\emptyset$.

Well formed environments:

$$\vdash \emptyset$$

$$\frac{\vdash E \quad x \notin dom(E)}{\vdash E, x : T}$$

$$\frac{\begin{array}{cc} \vdash E & n \notin dom(E) \\ E \vdash M_1 : T_1 \ldots E \vdash M_k : T_k & E \vdash N : R \end{array}}{\vdash E, n : T :: \{M_1, \ldots, M_k, n\}_N}$$

Environment lookups and subsumption:

$$\frac{E \vdash M : T \qquad T \sqsubseteq R}{E \vdash M : R}$$

$$\frac{\vdash E \qquad x : T \in E}{E \vdash x : T}$$

$$\frac{\vdash E \qquad n : T :: \{M_1, \ldots, M_k, n\}_N \in E}{E \vdash n : T}$$

$$\frac{\vdash E}{E \vdash 0 : \mathsf{public}}$$

$$\frac{E \vdash M : T}{E \vdash \mathsf{succ}\,(M) : T}$$

$$\frac{E \vdash M : T \qquad E \vdash N : T}{E \vdash \langle M, N \rangle : T}$$

Encryption

$$\frac{E \vdash M_1 : T \quad \ldots \quad E \vdash M_k : T \quad E \vdash N : \mathsf{public} \quad T = \mathsf{public} \text{ if } k = 0}{E \vdash \{M_1, \ldots, M_k\}_N : T}$$

$$\frac{\begin{array}{ccc} E \vdash M_1 : \mathsf{secret} & E \vdash M_2 : \mathsf{any} & E \vdash M_3 : \mathsf{public} \\ E \vdash N : \mathsf{secret} & \multicolumn{2}{c}{n : T :: \{M_1, M_2, M_3, n\}_N \in E} \end{array}}{E \vdash \{M_1, M_2, M_3, n\}_N : \mathsf{public}}$$

$$\frac{E \vdash M : \mathsf{public} \quad E \vdash M_1 : \mathsf{public} \quad \ldots \quad E \vdash M_k : \mathsf{public} \quad E \vdash P}{E \vdash \mathsf{send}_M \langle M_1, \ldots, M_k \rangle ; P}$$

$$\frac{E \vdash M : \mathsf{secret} \quad E \vdash M_1 : \mathsf{secret} \quad E \vdash M_2 : \mathsf{any} \quad E \vdash M_3 : \mathsf{public} \quad E \vdash P}{E \vdash \mathsf{send}_M \langle M_1, M_2, M_3 \rangle ; P}$$

Only public data may be sent on public channels.

On secret channels, data is always sent in the standard format we have agreed upon.

We consider pairing as left-associative.

For example (M_1, M_2, M_3, M_4) is same as $((M_1, M_2), M_3, M_4)$

Similar rules for inputs.

$$\frac{E \vdash M : \mathsf{public} \quad E, x_1 : \mathsf{public}, \ldots, x_k : \mathsf{public} \vdash P}{E \vdash \mathsf{recv}_M(x_1, \ldots, x_k); P}$$

$$\frac{E \vdash M : \mathsf{secret} \quad E, x_1 : \mathsf{secret}, x_2 : \mathsf{any}, x_3 : \mathsf{public} \vdash P}{E \vdash \mathsf{recv}_M(x_1, x_2, x_3); P}$$

The appropriate class information for the input variables is added to the environment, and the new environment is used for typing the remaining process.

$$\frac{\vdash E}{E \vdash \mathsf{halt}}$$

$$\frac{E \vdash P \qquad E \vdash Q}{E \vdash P \mid Q}$$

$$\frac{E \vdash P}{E \vdash \mathsf{repeat}\ P}$$

$$\frac{E, n : T :: L \vdash P}{E \vdash \mathsf{new}\ n; P}$$

The newly created name can be chosen to be kept secret or can be revealed, and can be chosen to used as a confounder in some message.

$$\boxed{\dfrac{E \vdash M : T \qquad E \vdash N : R \qquad E \vdash P \qquad T, R \in \{\mathsf{public}, \mathsf{secret}\}}{E \vdash \mathsf{check}\ (M == N); P}}$$

Equality checks are not allowed on data of class any to prevent implicit information flow.

Example Consider $P \triangleq \mathsf{recv}_c(y); \mathsf{check}\ (x == y); \mathsf{send}_c\langle 0\rangle; \mathsf{halt}$ where x is the data whose secrecy we are interested in.

Secrecy of x is not maintained. $P[M/x]$ and $P[M'/x]$ are not equivalent for $M \neq M'$.

Consider test $(Q, \overline{d})$ where $Q \triangleq \mathsf{send}_c\langle M\rangle; \mathsf{recv}_c(z); \mathsf{send}_d\langle 0\rangle; \mathsf{halt}$.

$P[M/x] \mid Q$ passes the test:

$P[M/x] \mid Q \xrightarrow{\tau} \mathsf{check}\ (M = M); \mathsf{send}_c\langle 0\rangle; \mathsf{halt} \mid \mathsf{recv}_c(z); \mathsf{send}_d\langle 0\rangle; \mathsf{halt} \xrightarrow{\tau}$
$\mathsf{halt} \mid \mathsf{send}_d\langle 0\rangle; \mathsf{halt} \xrightarrow{\overline{d}} \langle 0\rangle(\mathsf{halt} \mid \mathsf{halt})$

$P[M'/x] \mid Q$ does not pass the test.

Similarly, case analysis on data of class any are disallowed.

$$\boxed{\begin{array}{c}\dfrac{E \vdash M : T \qquad E, x : T, y : T \vdash P \qquad T \in \{\mathsf{public}, \mathsf{secret}\}}{E \vdash \mathsf{let}\ (x, y) = M; P} \\[2em] \dfrac{E \vdash M : T \qquad E \vdash P \qquad E, x : T \vdash Q \qquad T \in \{\mathsf{secret}, \mathsf{public}\}}{E \vdash \mathsf{case}\ M\ \mathsf{of}\ 0 : P,\ \mathsf{succ}\ (x) : Q}\end{array}}$$

Decryption

$$\frac{E \vdash L : T \quad E \vdash N : \mathsf{public} \quad E, x_1 : T, \ldots, x_k : T \vdash P \quad T \in \{\mathsf{secret}, \mathsf{public}\}}{E \vdash \mathsf{case}\ L\ \mathsf{of}\ \{x_1, \ldots, x_k\}_N : P}$$

$$\frac{\begin{array}{c} E \vdash L : T \qquad E \vdash N : \mathsf{secret} \qquad T \in \{\mathsf{secret}, \mathsf{public}\} \\ E, x_1 : \mathsf{secret}, x_2 : \mathsf{any}, x_3 : \mathsf{public}, x_4 : \mathsf{any} \vdash P \end{array}}{E \vdash \mathsf{case}\ L\ \mathsf{of}\ \{x_1, x_2, x_3, x_4\}_N : P}$$

The confounder x_4 in the second rule is assumed to be of type any because we have no more information about it.

Typing implies noleak of information

Suppose

- $\vdash E$
- all variables in $dom(E)$ are of level any and all names in $dom(E)$ are of level public.
- $E \vdash P$
- P has free variables $x_1, \ldots, x_k$
- $fn(M_i), fn(M_i') \subseteq dom(E)$ for $1 \leq i \leq k$.

then $P[M_1/x_1, \ldots, M_k/x_k] \simeq P[M_1/x_1, \ldots, M_k/x_k]$

Well typed processes maintain secrecy of the free variables $(x_1, \ldots, x_k)$, i.e. they are not leaked.

Our previous example $P \triangleq \mathsf{recv}_c(y); \mathsf{check}\ (x == y); \mathsf{send}_c\langle 0\rangle; \mathsf{halt}$

We take $E \triangleq \{x : \mathsf{any}, c : \mathsf{public} :: \{n\}_0\}$. c is not meant to be used as a confounder, hence we have the dummy term $\{n\}_0$.

We have $\vdash E$.

In order to show $E \vdash P$ we need to find some T such that

$E, y : \mathsf{public} \vdash \mathsf{check}\ (x == y); \mathsf{send}_c\langle 0\rangle; \mathsf{halt}$.

But this is impossible because equality checks should not involve data of class any.

Hence the process doesn't type-check, as required.

Consider $P \triangleq \mathsf{new}\ K; \mathsf{new}\ m; \mathsf{new}\ n; \mathsf{send}_c\langle\{m, x, 0, n\}_K\rangle; \mathsf{halt}$.

We take $E \triangleq \{x : \mathsf{any}, c : \mathsf{public} :: \{n\}_0\}$. We have $\vdash E$.

To show $E \vdash P$ we choose
$E' \triangleq E, K : \mathsf{secret} :: \{K\}_0, m : \mathsf{secret} :: \{m\}_0, n : \mathsf{secret} :: \{m, x, 0, n\}_K$
and show that $E' \vdash \mathsf{send}_c\langle\{m, x, 0, n\}_K\rangle; \mathsf{halt}$.

This is ok because $E' \vdash m : \mathsf{secret}$, $E' \vdash x : \mathsf{any}$, $E' \vdash 0 : \mathsf{public}$, $E' \vdash n : \mathsf{secret}$, $E' \vdash K : \mathsf{secret}$ and $E' \vdash \mathsf{halt}$.