FROM RESEARCH TO INDUSTRY

cea tech

C2TLA+: A translator from C to TLA+

Amira METHNI (CEA/CNAM)
Matthieu LEMERRE (CEA) & Belgacem BEN HEDIA (CEA)
Serge HADDAD (ENS Cachan) & Kamel BARKAOUI (CNAM)

www.cea.fr

June 3, 2014

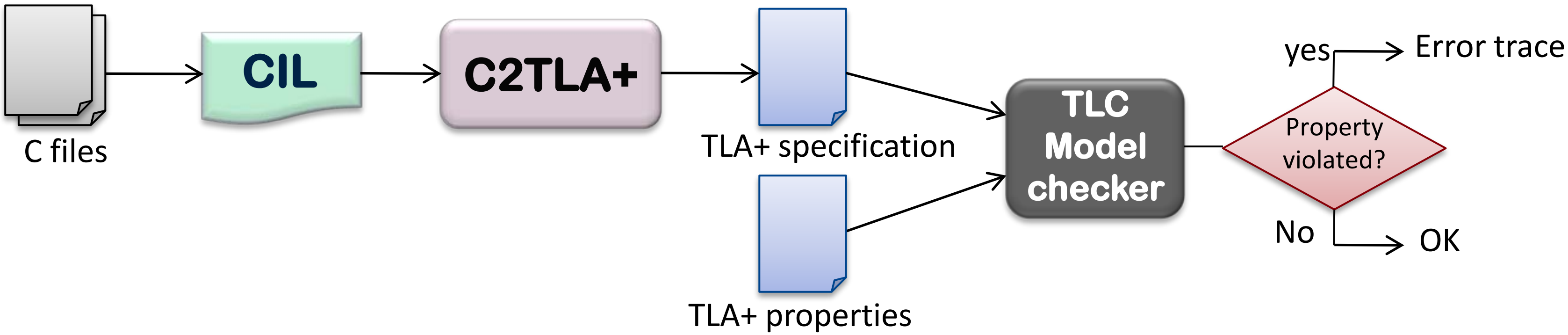
leti & list

Outline

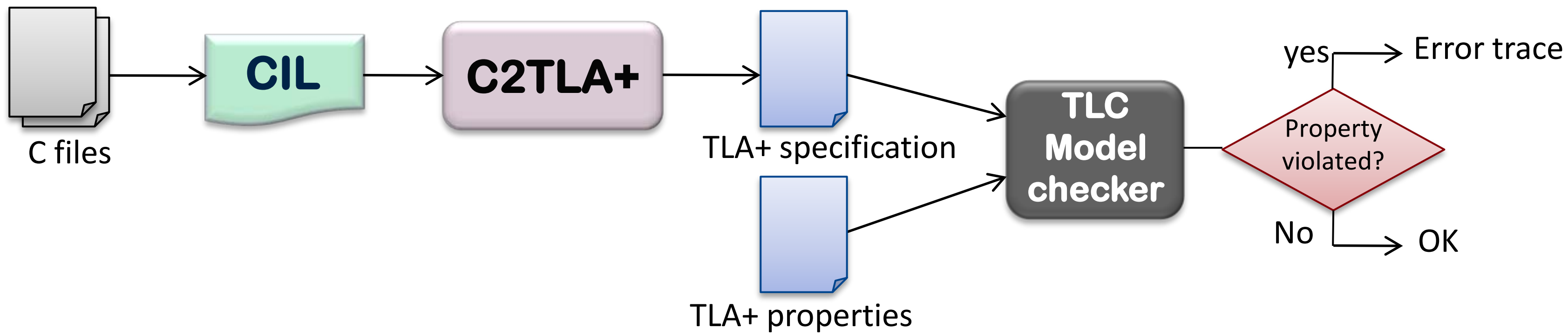
- Introduction
- General approach
- Translation from C to TLA+
  - Memory model
  - Expressions
  - Intra-procedural control flow
  - Inter-procedural control flow
  - Generating specification
  - Examples of properties
- Conclusion

- Context
  - C is a low level language
  - Programs are concurrent
    - Verifying C code is challenging (presence of pointer, pointer arithmetic's...)
- Motivation
  - Verifying an implementation model.
  - Guaranteeing the absence of certain classes of errors.
- Method
  - Automatically translate a TLA+ specification from input C codes.
  - Using automated tools to verify concurrent C programs against a set of safety and liveness properties.

General approach



General approach



- **CIL** (**C I**ntermediate **Language**) is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs.
  - Some of CIL's simplifications:
    - All forms of loops (while, for and do) are compiled internally as a single while(1) looping construct with explicit goto statements (for termination.)

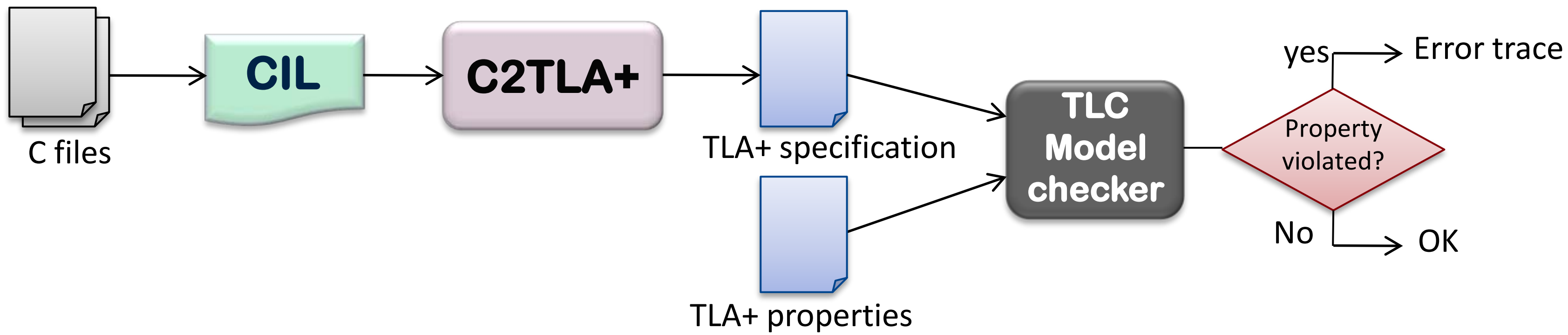
C

```
while (x<10){
 if (x == 8)
    continue;
 x++;
}
```

CIL

```
while (1) {
  while_continue: /* internal */ ;
  if (! (x < 10))
   { goto while_break; }
  if (x == 8)
   { goto while_continue; }
  x ++;
}
while_break: /* internal */ ;
```

General approach



- **CIL** (**C I**ntermediate **Language**) is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs.
  - Some of CIL's simplifications:
    - Expressions that contain side-effects are separated into statements.

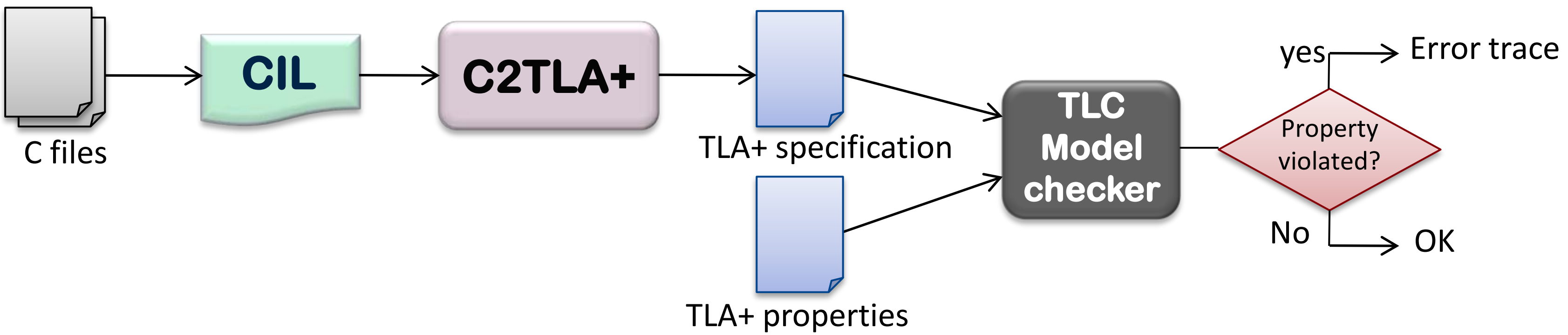
C

```
return(y ++ + f(x++));
```

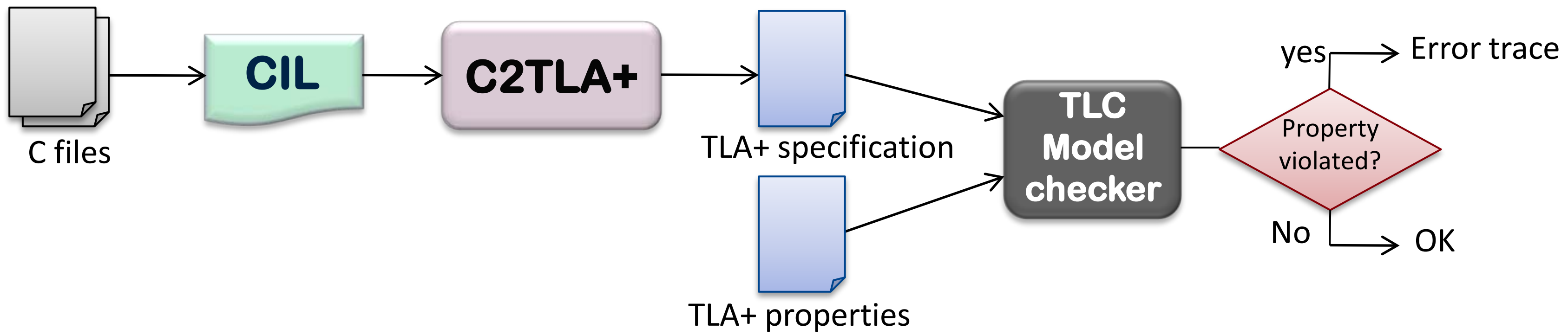
CIL

```
int tmp = y;
y ++;
int tmp_0 = x;
x ++;
int tmp_1 = f(tmp_0);
int __retres = tmp + tmp_1;
return (__retres);
```

General approach

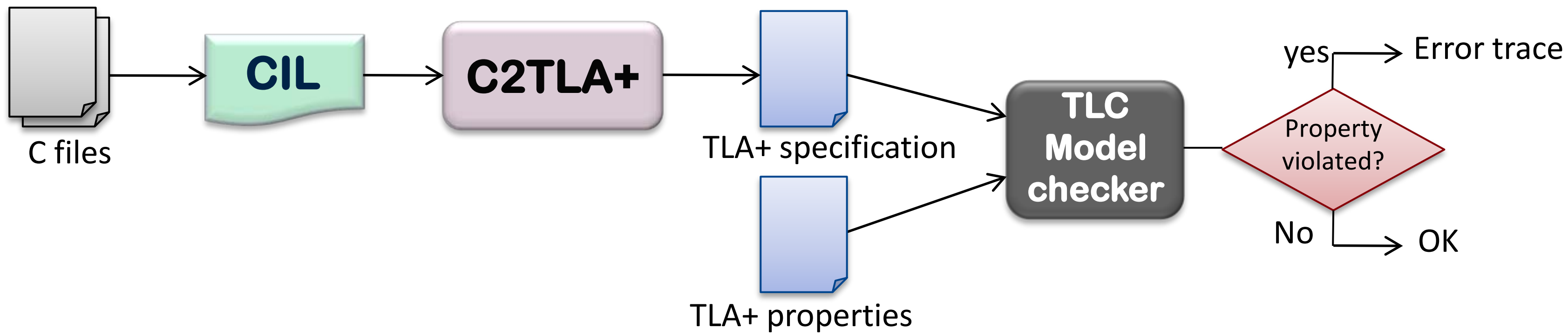


General approach



- Expressions: pointers, pointer arithmetic, referencing, dereferencing (&), array indexing, structure members (.), arithmetic (*, +, -, %, /), relational (>, >=, <, <=, ==, !=) and logical (&&, ||, !) operators;
- Statements: assignment, conditions (`if`,`if/else`), loops (`for/do-while/while`), `goto`, `break`, `continue`, `return`;
- Data types: integers (`int`), structures (`struct`), enumerations (`enum`);
- Value-returning function of `int` or pointer type;
- Recursion.
- C2TLA does not support: functions pointer, dynamic memory allocation and assignments of structures types.

General approach



- **Using TLC to verify properties**
  - Safety
    - Problems because of pointers and arrays (dereferencing null pointer).
    - Invariants over variables values.
    - Mutual exclusion.
  - Liveness
    - Termination.
    - Starvation-freedom (each waiting process will eventually enter its critical section).

Memory Model

- Concurrent program consists in several interleaved sequences of operations called processes (corresponding to threads in C).
- C2TLA+ attributes a unique identifier to each process, and defines the constant Procs to be the set of all process identifiers.

Memory Model

- Four memory regions:
  1) A region that contains global (and static) variables, called ***mem***.
     - Shared by processes.
     - Modeled by an array (function).

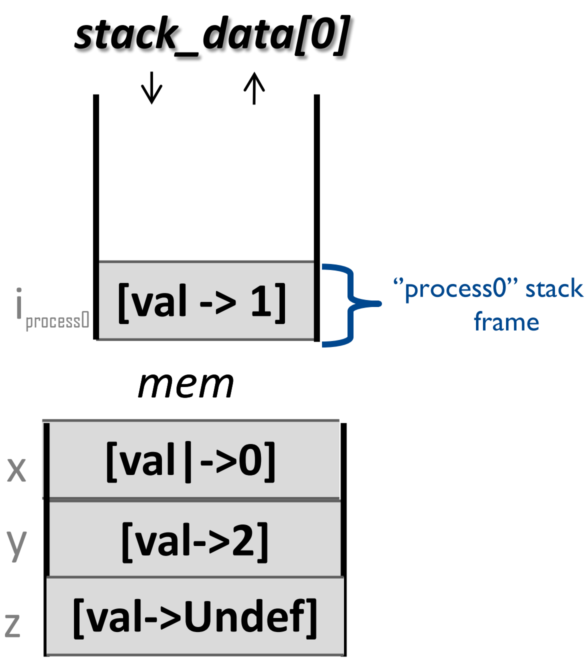
```
int x = 0;
int y = 2;
int z;

int max(int a,int b)
{if (a>=b)
    return a;
 else return b;}

int process0(){
    int i = 1;
    x = x + i;
    y = max(x,y);
    x = y + 1;
    return x;  }

void process1(){
    int j = 0;
    x = max(x,y); }
```

mem

x	[val|->0]
y	[val->2]
z	[val->Undef]

Memory Model

- Four memory regions:
  - **2)** A region contains local variables and function parameters, called *stack_data* .
    - This region is modeled by an array of sequences and is composed of stack frames.
    - Each stack frame corresponds to a call to a function which has not yet terminated with a return.




```
int x = 0;
int y = 2;
int z;

int max(int a,int b)
{if (a>=b)
    return a;
 else return b;}

int process0(){
    int i = 1;
    x = x + i;
    y = max(x,y);
    x = y + 1;
    return x;  }

void process1(){
    int j = 0;
    x = max(x,y); }
```

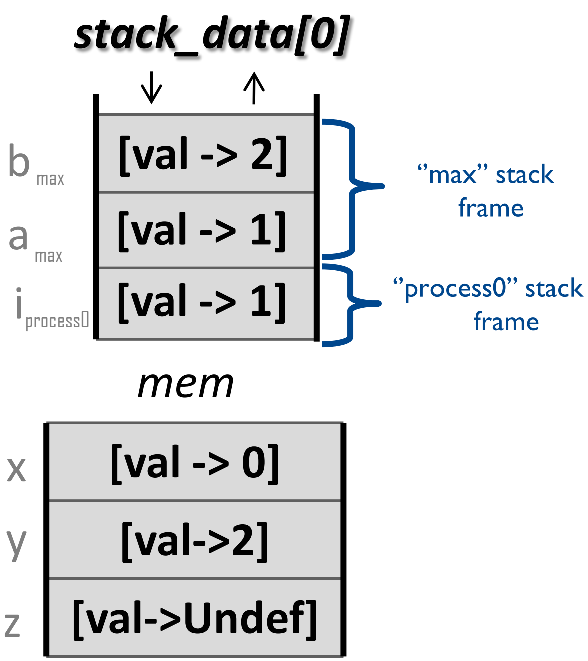
Memory Model

- Four memory regions:

2) A region contains local variables and function parameters, called ***stack_data*** .

- This region is modeled by an array of sequences and is composed of stack frames.
- Each stack frame corresponds to a call to a function which has not yet terminated with a return.

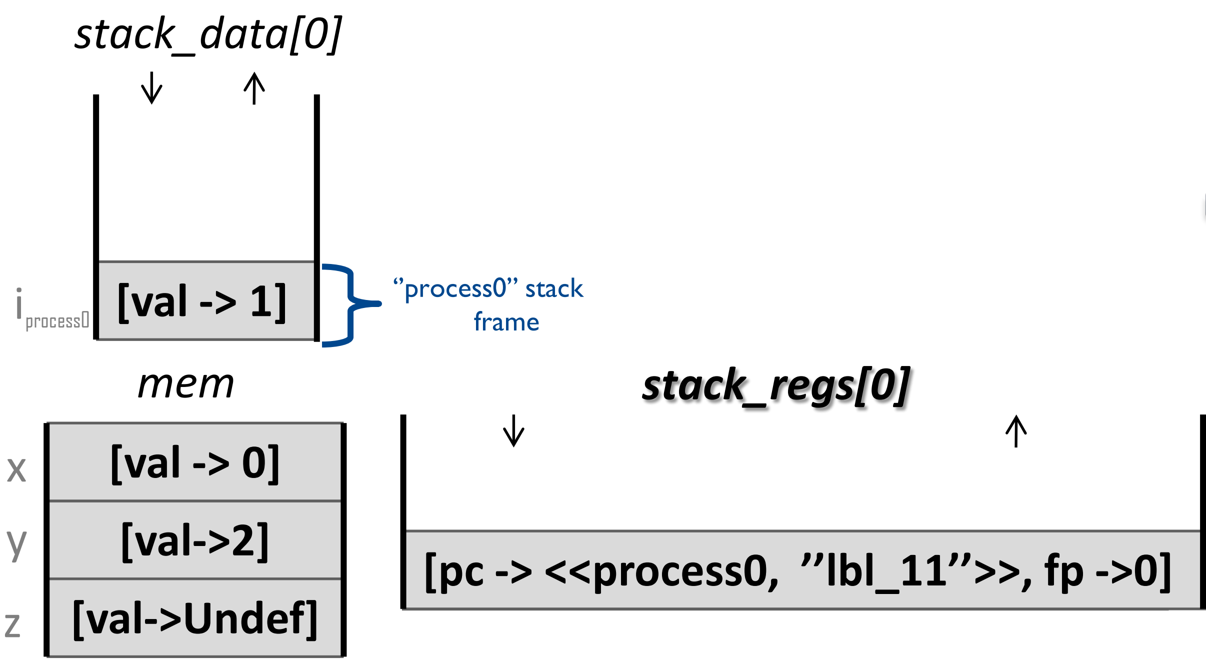
```
int x = 0;
int y = 2;
int z;

int max(int a,int b)
{if (a>=b)
    return a;
 else return b;}

int process0(){
    int i = 1;
    x = x + i;
    y = max(x,y);
    x = y + 1;
    return x;   }

void process1(){
    int j = 0;
    x = max(x,y); }
```



Memory Model

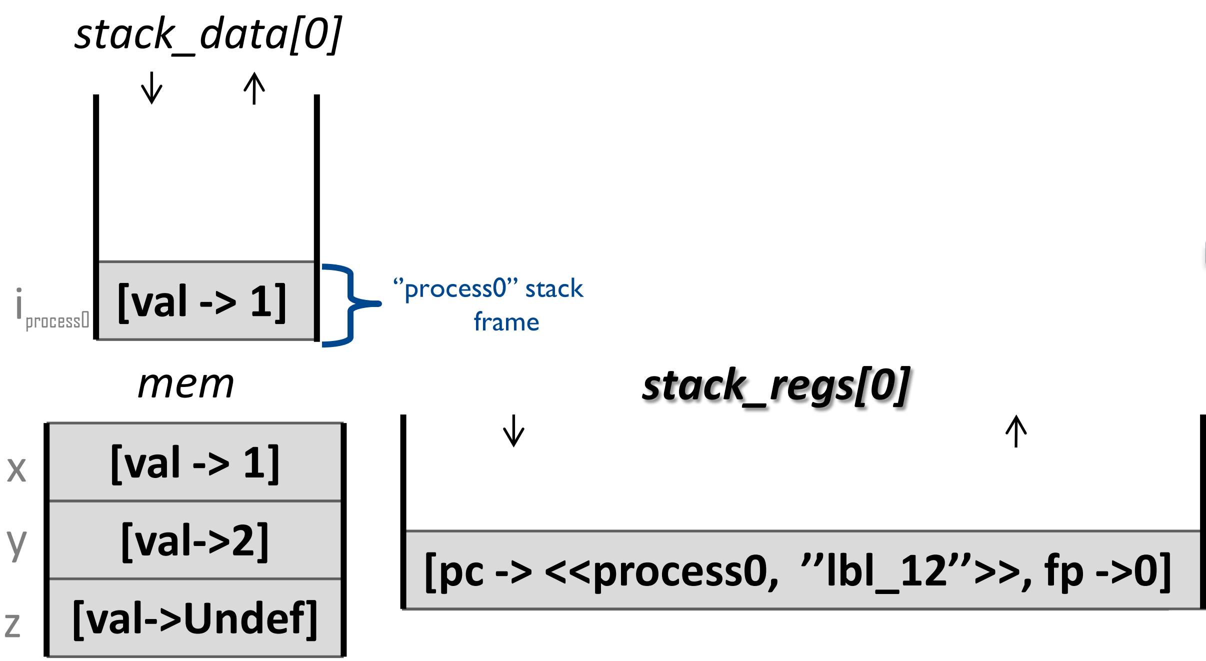
- Four memory regions:
  3) A region that stores the program counter of each process (***stack_regs***).
  - It associates to each process a stack of records.
  - Each record contains two fields:
    - ***pc***, the program counter, represented by a tuple function <<name, label>> (Labels values are given by CIL).
    - ***fp***, the frame pointer, contains the base offset of the current stack frame.

```
int x = 0;
int y = 2;
int z;

int max(int a,int b)
{if (a>=b)
    return a;
 else return b;}

int process0(){
    int i = 1;
    x = x + i;
    y = max(x,y);
    x = y + 1;
    return x;  }

void process1(){
    int j = 0;
    x = max(x,y); }
```



Memory Model

- Four memory regions:
  - **3)** A region that stores the program counter of each process (***stack_regs***).
    - It associates to each process a stack of records.
    - Each record contains two fields:
      - *pc*, the program counter, represented by a tuple function <<name, label>> (Labels values are given by CIL).
      - *fp*, the frame pointer, contains the base offset of the current stack frame.



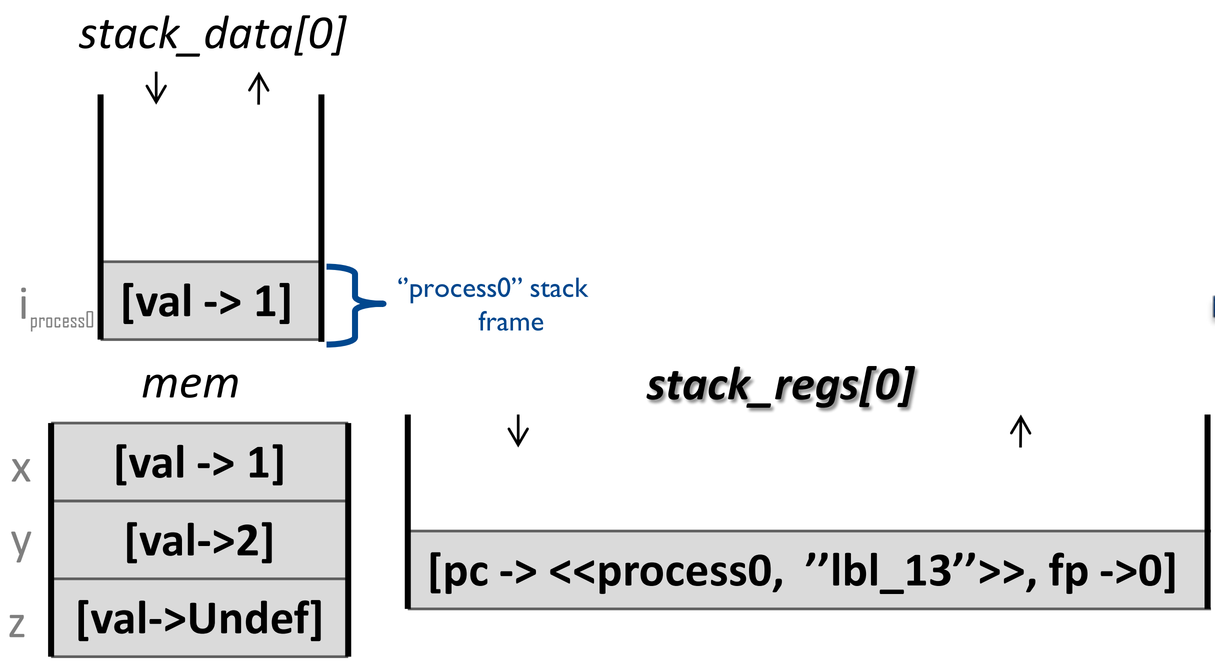

```
int x = 0;
int y = 2;
int z;

int max(int a,int b)
{if (a>=b)
    return a;
 else return b;}

int process0(){
    int i = 1;
    x = x + i;
    y = max(x,y);
    x = y + 1;
    return x;   }

void process1(){
    int j = 0;
    x = max(x,y); }
```

Memory Model

- Four memory regions:

  3) A region that stores the program counter of each process (***stack_regs***).

  - It associates to each process a stack of records.
  - Each record contains two fields:
    - *pc*, the program counter, represented by a tuple function <<name, label>> (Labels values are given by CIL).
    - *fp*, the frame pointer, contains the base offset of the current stack frame.



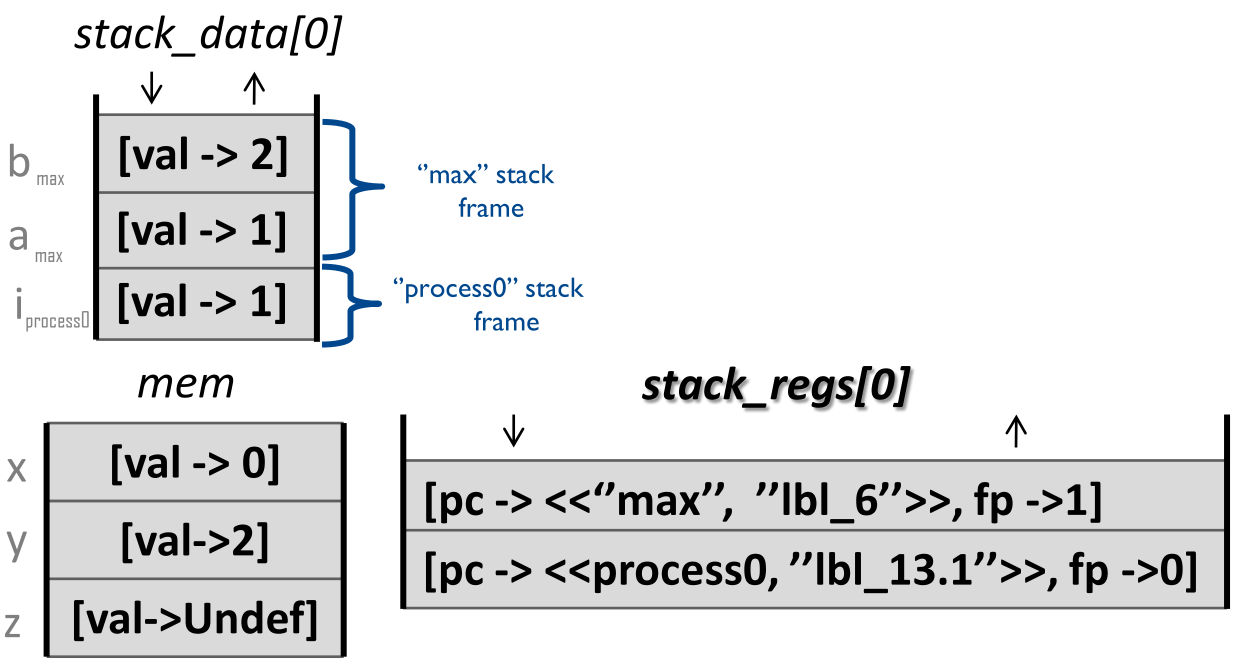

```
int x = 0;
int y = 2;
int z;

int max(int a,int b)
{if (a>=b)
    return a;
 else return b;}

int process0(){
    int i = 1;
    x = x + i;
    y = max(x,y);
    x = y + 1;
    return x;   }

void process1(){
    int j = 0;
    x = max(x,y); }
```

- Four memory regions:

  3) A region that stores the program counter of each process (***stack_regs***).

  - It associates to each process a stack of records.
  - Each record contains two fields:
    - *pc*, the program counter, represented by a tuple function <<name, label>> (Labels values are given by CIL).
    - *fp*, the frame pointer, contains the base offset of the current stack frame.

```
1 -  int x = 0;
2 -  int y = 2;
3 -  int z;
4 -
5 -  int max(int a,int b)
6 -  {if (a>=b)
        return a;
7 -
8 -   else return b;}
9 -
10 - int process0(){
11 -     int i = 1;
12 -     x = x + i;
13 -     y = max(x,y);
14 -     x = y + 1;
15 -     return x;  }
16 -
17 - void process1(){
18 -     int j = 0;
19 -     x = max(x,y); }
```

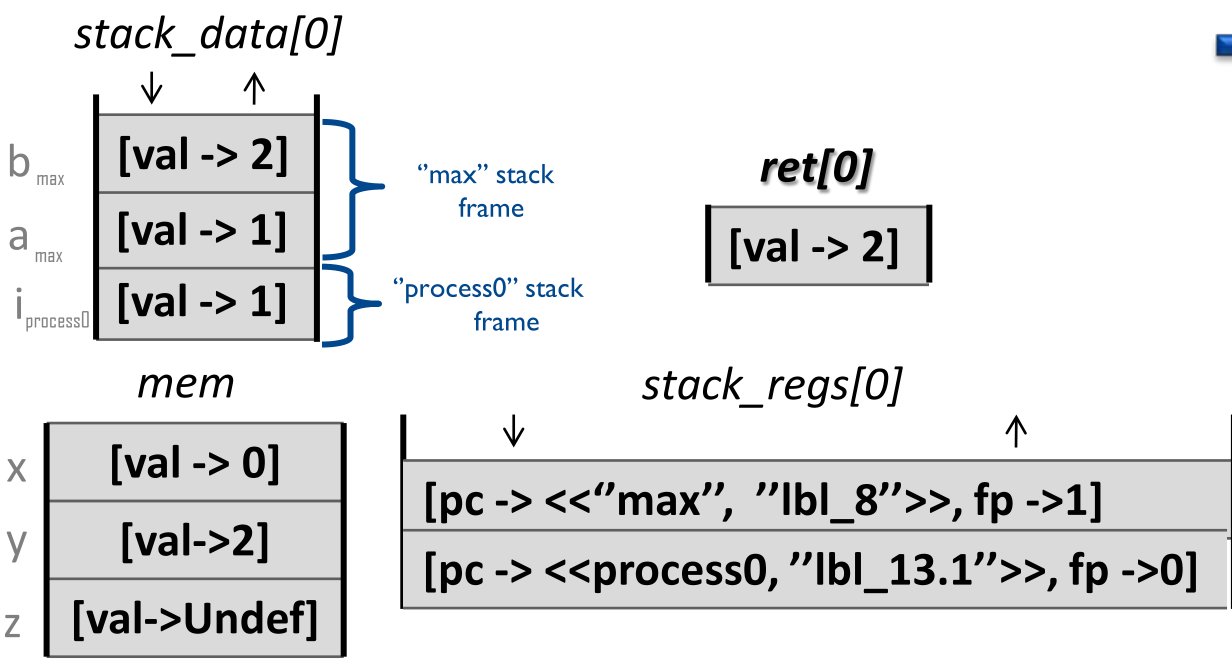


Memory Model

- Four memory regions:

4) A region contains the values returned by processes, called ***ret***.

- This region is modeled by an array and indexed by process identifier.




```
int x = 0;
int y = 2;
int z;

int max(int a,int b)
{if (a>=b)
    return a;
 else return b;}

int process0(){
    int i = 1;
    x = x + i;
    y = max(x,y);
    x = y + 1;
    return x;   }

void process1(){
    int j = 0;
    x = max(x,y); }
```

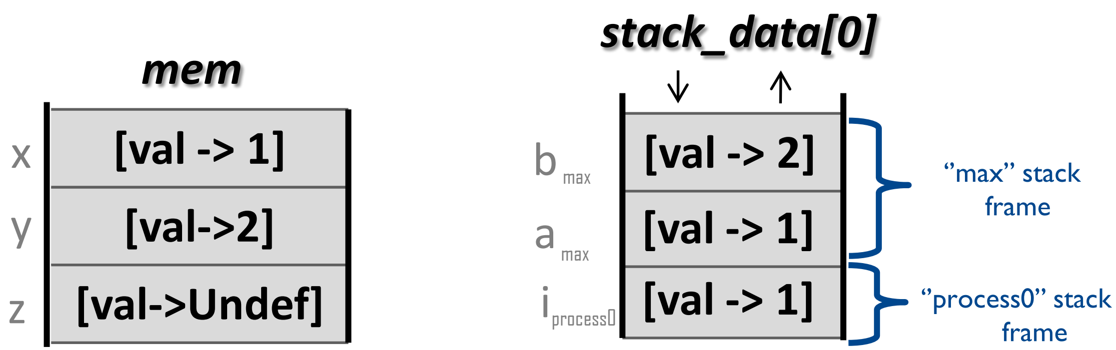
Memory Model

- C2TLA+ maps each C variable to unique TLA+ constant (address) modeled by a record with two fields :
  - *loc* : memory region (*mem* or *stack_data)*.
  - *off* : offset in the considered memory region.
- Example

$$Addr\_x \triangleq [loc \mapsto \text{"mem"}, offs \mapsto 0]$$

$$Addr\_process0\_i \triangleq [loc \mapsto \text{"stack\_data"}, offs \mapsto 0]$$

```
int x = 0;
int y = 2;
int z;
int process0(){
   int i = 1;
   x = x + i;
   y = max(x,y);
   x = y + 1;
   return x;
}
void process1(){
   int j = 0;
   x = max(x,y);
}
```



Memory Model

- C2TLA+ maps each C variable to unique TLA+ constant modeled by a record with two fields :
  - *loc* : memory region (*mem* ou *stack_data)*.
  - *off* : offset in the considered memory region.

- **Loading operation**
  - A *lvalue* is evaluates to an address and which refers to a region of storage.
  - Accessing the value stored in this region is performed using the TLA+ operator *load()*.

$$load(id,\ ptr) \triangleq \text{IF } ptr.loc = \text{“mem”} \text{ THEN } mem[ptr.offs] \\ \text{ELSE } stack\_data[id][Head(stack\_regs[id]).fp + ptr.offs]$$

Memory Model

- C2TLA+ maps each C variable to unique TLA+ constant modeled by a record with two fields :
  - *loc* : memory region (*mem* ou *stack_data)*.
  - *off* : offset in the considered memory region.
- **Loading operation**
- **Assignment operation**

$$
\begin{array}{l}
store(id,\ ptr,\ value)\ \triangleq \\
\lor\ \ \land\ ptr.loc = \text{“mem”} \\
\quad\ \ \land\ mem' = [mem\ \text{EXCEPT}\ ![ptr.offs] = value] \\
\quad\ \ \land\ \text{UNCHANGED}\ stack\_data \\
\lor\ \ \land\ ptr.loc = \text{“stack\_data”} \\
\quad\ \ \land\ stack\_data' = [stack\_data\ \text{EXCEPT}\ ![id][Head(stack\_regs[id]).fp + ptr.offs] = value] \\
\quad\ \ \land\ \text{UNCHANGED}\ mem
\end{array}
$$

- Example

The statement `i = 1;` is translated into TLA+ as $store(id, Addr\_process0\_i, [val \mapsto 1])$

Expressions

- **Arrays**
  - Accessing an array element in C2TLA+ requires computing the offset using the size of the elements, the index and the base address of the array.
  - Example: accessing to `z[a]` is translated as

$$load(id, [loc \mapsto Addr\_z.loc, offs \mapsto (Addr\_z.offs + (load(id, Addr\_a) * Size\_of\_int))])$$

- **Pointer arithmetic's and structure member**
  - The same kind computation is used to perform pointer arithmetics.
  - Similarly, accessing a structure member is achieved by shifting the base address of the structure with the constant accumulated size of all previous members.
  - Example: accessing to `student.name` is translated as

$$load(id, [loc \mapsto Addr\_student.loc, offs \mapsto (Addr\_student.offs + Offset\_student\_name)])$$

Intra-procedural Control Flow

Function definition

- Each C function definition is translated into an operator with the process identifier *id* as argument.
- A C statement is translated into the conjunction of actions that are done simultaneously.
- The function body is translated into the disjunction of the translation of each statement it contains.

Intra-procedural Control Flow

Function definition

- Example

```
...
int process0(){
   int i = 1;
   x = x + i;
   return x;
}
```

- Example

$$process0(id) \triangleq \lor \land Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_1”} \rangle$$
$$\land\ store(id, Addr\_process0\_i, [val \mapsto 1])$$
$$\land\ stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_2”} \rangle,$$
$$fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])]$$
$$\land\ \text{UNCHANGED } \langle ret \rangle$$

```
...
int process0(){
 1  int i = 1;
 2  x = x + i;
 3  return x;
}
```

- Example

$$
\begin{aligned}
process0(id) \triangleq\ & \lor \land Head(stack\_regs[id]).pc = \langle \text{"process0"}, \text{"lbl\_1"} \rangle \\
& \quad \land store(id, Addr\_process0\_i, [val \mapsto 1]) \\
& \quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"process0"}, \text{"lbl\_2"} \rangle, \\
& \qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
& \quad \land \text{UNCHANGED } \langle ret \rangle \\
& \lor \land Head(stack\_regs[id]).pc = \langle \text{"process0"}, \text{"lbl\_2"} \rangle \\
& \quad \land store(id, Addr\_x, plus(load(id, Addr\_x), load(id, Addr\_process0\_i))) \\
& \quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"process0"}, \text{"lbl\_3"} \rangle, \\
& \qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
& \quad \land \text{UNCHANGED } \langle ret \rangle
\end{aligned}
$$

```
...
int process0(){
  int i = 1;
  x = x + i;
  return x;
}
```

- Example

$$
\begin{aligned}
process0(id) \triangleq\ & \lor \land Head(stack\_regs[id]).pc = \langle \text{"process0"}, \text{"lbl\_1"} \rangle \\
& \quad \land store(id, Addr\_process0\_i, [val \mapsto 1]) \\
& \quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"process0"}, \text{"lbl\_2"} \rangle, \\
& \qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
& \quad \land \text{UNCHANGED } \langle ret \rangle \\
& \lor \land Head(stack\_regs[id]).pc = \langle \text{"process0"}, \text{"lbl\_2"} \rangle \\
& \quad \land store(id, Addr\_x, plus(load(id, Addr\_x), load(id, Addr\_process0\_i))) \\
& \quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"process0"}, \text{"lbl\_3"} \rangle, \\
& \qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
& \quad \land \text{UNCHANGED } \langle ret \rangle \\
& \lor \land Head(stack\_regs[id]).pc = \langle \text{"process0"}, \text{"lbl\_3"} \rangle \\
& \quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = Tail(stack\_regs[id])] \\
& \quad \land stack\_data' = [stack\_data \text{ EXCEPT } ![id] = \\
& \qquad\qquad SubSeq(stack\_data[id], 1, Head(stack\_regs[id]).fp - 1)] \\
& \quad \land ret' = [ret \text{ EXCEPT } ![id] = load(id, Addr\_x)] \\
& \quad \land \text{UNCHANGED } \langle mem \rangle
\end{aligned}
$$

```
...
int process0(){
 1  int i = 1;
 2  x = x + i;
 3  return x;
}
```

Intra-procedural Control Flow

Jump statement

- The translation of `goto/break/continue` statements consists in updating *stack_regs[id]* to the successor statement.

```
lbl0 : if (x < 10)
  goto lbl1;
else goto lbl2;
lbl1: x ++;
goto lbl0;
lbl2: y = x;
...
```

Intra-procedural Control Flow

Jump statement

- The translation of `goto/break/continue` statements consists in updating ***stack_regs[id]*** to the successor statement.
- Example:

$$
\begin{aligned}
\lor\ & \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_1”} \rangle \\
& \land\ \text{IF } (lt((load(id, Addr\_x, ([val \mapsto 10])) \neq [val \mapsto 0]) \\
& \quad \text{THEN } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_2”} \rangle, \\
& \qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
& \quad \text{ELSE } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_3”} \rangle, \\
& \qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
& \land\ \text{UNCHANGED } \langle mem, stack\_data, ret \rangle
\end{aligned}
$$

```
lbl0 : if (x < 10)
  goto lbl1;
else goto lbl2;
lbl1: x ++;
goto lbl0;
lbl2: y = x;
...
```

Intra-procedural Control Flow

Jump statement

- The translation of `goto/break/continue` statements consists in updating ***stack_regs[id]*** to the successor statement.
- Example:

```
lbl0 : if (x < 10)
  goto lbl1;
else goto lbl2;
lbl1: x ++;
goto lbl0;
lbl2: y = x;
...
```

$$
\begin{array}{l}
\lor \ \land Head(stack\_regs[id]).pc = \langle \text{"process0"}, \text{"lbl\_1"} \rangle \\
\quad \land \text{IF } (lt((load(id, Addr\_x, ([val \mapsto 10])) \neq [val \mapsto 0]) \\
\qquad \text{THEN } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"process0"}, \text{"lbl\_2"} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\qquad \text{ELSE } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"process0"}, \text{"lbl\_3"} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor \ \land Head(stack\_regs[id]).pc = \langle \text{"process0"}, \text{"lbl\_2"} \rangle \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"process0"}, \text{"lbl\_4"} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, stack\_data, ret \rangle
\end{array}
$$

Intra-procedural Control Flow

Jump statement

- The translation of `goto/break/continue` statements consists in updating ***stack_regs[id]*** to the successor statement.
- Example:

```
lbl0 : if (x < 10)
  goto lbl1;
else goto lbl2;
lbl1: x ++;
goto lbl0;
lbl2: y = x;
...
```

$$
\begin{array}{l}
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_1”} \rangle \\
\quad \land\ \text{IF}\ (lt((load(id,\ Addr\_x,\ ([val \mapsto 10])) \neq [val \mapsto 0]) \\
\qquad \text{THEN}\ stack\_regs' = [stack\_regs\ \text{EXCEPT}\ ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_2”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\qquad \text{ELSE}\ stack\_regs' = [stack\_regs\ \text{EXCEPT}\ ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_3”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land\ \text{UNCHANGED}\ \langle mem,\ stack\_data,\ ret \rangle \\
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_2”} \rangle \\
\quad \land\ stack\_regs' = [stack\_regs\ \text{EXCEPT}\ ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_4”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land\ \text{UNCHANGED}\ \langle mem,\ stack\_data,\ ret \rangle \\
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_3”} \rangle \\
\quad \land\ stack\_regs' = [stack\_regs\ \text{EXCEPT}\ ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_6”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land\ \text{UNCHANGED}\ \langle mem,\ stack\_data,\ ret \rangle
\end{array}
$$

...

Intra-procedural Control Flow

Jump statement

- The translation of `goto/break/continue` statements consists in updating ***stack_regs[id]*** to the successor statement.
- Example:

```
lbl0 : if (x < 10)
  goto lbl1;
else goto lbl2;
lbl1: x ++;
goto lbl0;
lbl2: y = x;
...
```

$$
\begin{array}{l}
\lor \ \land \ Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_1”} \rangle \\
\quad \land \text{ IF } (lt((load(id, Addr\_x, ([val \mapsto 10])) \neq [val \mapsto 0]) \\
\qquad \text{THEN } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_2”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\qquad \text{ELSE } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_3”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{ UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor \ \land \ Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_2”} \rangle \\
\quad \land \ stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_4”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{ UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor \ \land \ Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_3”} \rangle \\
\quad \land \ stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_6”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{ UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor \ \land \ Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_4”} \rangle \\
\quad \land \ store(id, Addr\_x, plus(load(id, Addr\_x), [val \mapsto 1])) \\
\quad \land \ stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_5”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{ UNCHANGED } \langle ret \rangle
\end{array}
$$

…

Intra-procedural Control Flow

Jump statement

- The translation of `goto/break/continue` statements consists in updating ***stack_regs[id]*** to the successor statement.
- Example:

```
lbl0 : if (x < 10)
  goto lbl1;
else goto lbl2;
lbl1: x ++;
goto lbl0;
lbl2: y = x;
...
```

$$
\begin{array}{l}
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_1”} \rangle \\
\quad \land\ \text{IF } (lt((load(id, Addr\_x), ([val \mapsto 10])) \neq [val \mapsto 0]) \\
\qquad \text{THEN } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_2”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\qquad \text{ELSE } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_3”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land\ \text{UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_2”} \rangle \\
\quad \land\ stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_4”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land\ \text{UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_3”} \rangle \\
\quad \land\ stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_6”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land\ \text{UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_4”} \rangle \\
\quad \land\ store(id, Addr\_x, plus(load(id, Addr\_x), [val \mapsto 1])) \\
\quad \land\ stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_5”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land\ \text{UNCHANGED } \langle ret \rangle \\
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”}, \text{“lbl\_5”} \rangle \\
\quad \land\ stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”}, \text{“lbl\_1”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land\ \text{UNCHANGED } \langle mem, stack\_data, ret \rangle
\end{array}
$$

...

Intra-procedural Control Flow

Jump statement

- The translation of `goto/break/continue` statements consists in updating ***stack_regs[id]*** to the successor statement.
- Example:

```
lbl0 : if (x < 10)
  goto lbl1;
else goto lbl2;
lbl1: x ++;
goto lbl0;
lbl2: y = x;
...
```

$$
\begin{array}{l}
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”},\ \text{“lbl\_1”} \rangle \\
\quad \land\ \text{IF } (lt((load(id,\ Addr\_x,\ ([val \mapsto 10])) \neq [val \mapsto 0]) \\
\quad\quad \text{THEN } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”},\ \text{“lbl\_2”} \rangle, \\
\quad\quad\quad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad\quad \text{ELSE } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”},\ \text{“lbl\_3”} \rangle, \\
\quad\quad\quad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land\ \text{UNCHANGED } \langle mem,\ stack\_data,\ ret \rangle \\
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”},\ \text{“lbl\_2”} \rangle \\
\quad \land\ stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”},\ \text{“lbl\_4”} \rangle, \\
\quad\quad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”},\ \text{“lbl\_3”} \rangle \\
\quad \land\ stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”},\ \text{“lbl\_6”} \rangle, \\
\quad\quad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land\ \text{UNCHANGED } \langle mem,\ stack\_data,\ ret \rangle \\
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”},\ \text{“lbl\_4”} \rangle \\
\quad \land\ store(id,\ Addr\_x,\ plus(load(id,\ Addr\_x),\ [val \mapsto 1])) \\
\quad \land\ stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”},\ \text{“lbl\_5”} \rangle, \\
\quad\quad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land\ \text{UNCHANGED } \langle ret \rangle \\
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”},\ \text{“lbl\_5”} \rangle \\
\quad \land\ stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”},\ \text{“lbl\_1”} \rangle, \\
\quad\quad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land\ \text{UNCHANGED } \langle mem,\ stack\_data,\ ret \rangle \\
\lor\ \land\ Head(stack\_regs[id]).pc = \langle \text{“process0”},\ \text{“lbl\_6”} \rangle \\
\quad \land\ store(id,\ Addr\_y,\ load(id,\ Addr\_x)) \\
\quad \land\ stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“process0”},\ \text{“lbl\_7”} \rangle, \\
\quad\quad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land\ \text{UNCHANGED } \langle ret \rangle \\
\ldots
\end{array}
$$

Intra-procedural Control Flow

Iteration statement

- All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.
- C Example:

```
while (x!=10) {
 x ++;
}
```

C code

Intra-procedural Control Flow

Iteration statement

- All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.
- C Example:

```
while (1) {
 if (! (x != 10))
 { goto while_0_break; }
 x ++;
}
while_0_break: ;
```

Normalized code

Intra-procedural Control Flow

Iteration statement

- All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.
- C Example:

$$
\begin{array}{l}
\lor \land Head(stack\_regs[id]).pc = \langle \text{“f1”}, \text{“lbl\_1”} \rangle \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“f1”}, \text{“lbl\_2”} \rangle, fp \\
\qquad \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, stack\_data, ret \rangle
\end{array}
$$

```
while (1) {
 if (! (x != 10))
 { goto while_0_break; }
 x ++;
}
while_0_break: ;
```

Normalized code

Intra-procedural Control Flow

Iteration statement

- All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.
- C Example:

```
while (1) {
 if (! (x != 10))
 { goto while_0_break; }
 x ++;
}
while_0_break: ;
```

$$
\begin{array}{l}
\lor \land Head(stack\_regs[id]).pc = \langle \text{“f1”}, \text{“lbl\_1”} \rangle \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“f1”}, \text{“lbl\_2”} \rangle, fp \\
\qquad \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor \land Head(stack\_regs[id]).pc = \langle \text{“f1”}, \text{“lbl\_2”} \rangle \\
\quad \land \text{IF } (ne((load(id, [loc \mapsto Addr\_x.loc, offs \mapsto Addr\_x.offs])), ([val \mapsto 10])) \neq [val \mapsto 0]) \\
\qquad \text{THEN } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“f1”}, \text{“lbl\_4”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\qquad \text{ELSE } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“f1”}, \text{“lbl\_3”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor \land Head(stack\_regs[id]).pc = \langle \text{“f1”}, \text{“lbl\_6”} \rangle \\
\quad \land store(id, Addr\_x, plus(load(id, Addr\_x), [val \mapsto 1])) \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“f1”}, \text{“lbl\_1”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle ret \rangle \\
\lor \land Head(stack\_regs[id]).pc = \langle \text{“f1”}, \text{“lbl\_3”} \rangle \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“f1”}, \text{“lbl\_6”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, stack\_data, ret \rangle
\end{array}
$$

Intra-procedural Control Flow

Iteration statement

- All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.
- C Example:

```c
while (1) {
 if (! (x != 10))
 { goto while_0_break; }
 x ++;
}
while_0_break: ;
```

Normalized code

$$
\begin{array}{l}
\lor \land Head(stack\_regs[id]).pc = \langle \text{"f1"}, \text{"lbl\_1"} \rangle \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_2"} \rangle, fp \\
\qquad \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor \land Head(stack\_regs[id]).pc = \langle \text{"f1"}, \text{"lbl\_2"} \rangle \\
\quad \land \text{IF } (ne((load(id, [loc \mapsto Addr\_x.loc, offs \mapsto Addr\_x.offs])), ([val \mapsto 10])) \neq [val \mapsto 0]) \\
\qquad \text{THEN } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_4"} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\qquad \text{ELSE } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_3"} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor \land Head(stack\_regs[id]).pc = \langle \text{"f1"}, \text{"lbl\_3"} \rangle \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_6"} \rangle, \\
\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, stack\_data, ret \rangle
\end{array}
$$

Intra-procedural Control Flow

Iteration statement

- All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.
- C Example:

```
while (1) {
 if (! (x != 10))
 { goto while_0_break; }
 x ++;
}
while_0_break: ;
```

Normalized code

$$
\begin{array}{l}
\lor \land Head(stack\_regs[id]).pc = \langle \text{"f1"}, \text{"lbl\_1"} \rangle \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_2"} \rangle, fp \\
\qquad \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor \land Head(stack\_regs[id]).pc = \langle \text{"f1"}, \text{"lbl\_2"} \rangle \\
\quad \land \text{IF } (ne((load(id, [loc \mapsto Addr\_x.loc, offs \mapsto Addr\_x.offs])), ([val \mapsto 10])) \neq [val \mapsto 0]) \\
\qquad \text{THEN } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_4"} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\qquad \text{ELSE } stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_3"} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor \land Head(stack\_regs[id]).pc = \langle \text{"f1"}, \text{"lbl\_3"} \rangle \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_6"} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, stack\_data, ret \rangle \\
\lor \land Head(stack\_regs[id]).pc = \langle \text{"f1"}, \text{"lbl\_4"} \rangle \\
\quad \land store(id, Addr\_x, plus(load(id, Addr\_x), [val \mapsto 1])) \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_1"} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle ret \rangle
\end{array}
$$

Intra-procedural Control Flow

Iteration statement

- All loops in C are normalized by CIL as a single while(1) looping construct that we translate like other jump statements.
- C Example:

```c
while (1) {
 if (! (x != 10))
 { goto while_0_break; }
 x ++;
}
while_0_break: ;
```

Normalized code

$$
\begin{array}{l}
\lor \ \land \ Head(stack\_regs[id]).pc = \langle \text{"f1"}, \text{"lbl\_1"} \rangle \\
\quad \land \ stack\_regs' = [stack\_regs \ \text{EXCEPT} \ ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_2"} \rangle, fp \\
\qquad \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \ \text{UNCHANGED} \ \langle mem, stack\_data, ret \rangle \\
\lor \ \land \ Head(stack\_regs[id]).pc = \langle \text{"f1"}, \text{"lbl\_2"} \rangle \\
\quad \land \ \text{IF} \ (ne((load(id, [loc \mapsto Addr\_x.loc, offs \mapsto Addr\_x.offs])), ([val \mapsto 10])) \neq [val \mapsto 0]) \\
\qquad \text{THEN} \ stack\_regs' = [stack\_regs \ \text{EXCEPT} \ ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_4"} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\qquad \text{ELSE} \ stack\_regs' = [stack\_regs \ \text{EXCEPT} \ ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_3"} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \ \text{UNCHANGED} \ \langle mem, stack\_data, ret \rangle \\
\lor \ \land \ Head(stack\_regs[id]).pc = \langle \text{"f1"}, \text{"lbl\_3"} \rangle \\
\quad \land \ stack\_regs' = [stack\_regs \ \text{EXCEPT} \ ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_6"} \rangle, \\
\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \ \text{UNCHANGED} \ \langle mem, stack\_data, ret \rangle \\
\lor \ \land \ Head(stack\_regs[id]).pc = \langle \text{"f1"}, \text{"lbl\_4"} \rangle \\
\quad \land \ store(id, Addr\_x, plus(load(id, Addr\_x), [val \mapsto 1])) \\
\quad \land \ stack\_regs' = [stack\_regs \ \text{EXCEPT} \ ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_1"} \rangle, \\
\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \ \text{UNCHANGED} \ \langle ret \rangle \\
\lor \ \land \ Head(stack\_regs[id]).pc = \langle \text{"f1"}, \text{"lbl\_6"} \rangle \\
\quad \land \ stack\_regs' = [stack\_regs \ \text{EXCEPT} \ ![id] = \langle [pc \mapsto \langle \text{"f1"}, \text{"lbl\_7"} \rangle, \\
\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \ \text{UNCHANGED} \ \langle mem, stack\_data, ret \rangle
\end{array}
$$

Inter-procedural Control Flow

Function call

- A function call is translated in two actions :
  - The stack frame is pushed onto the ***stack_data[id]*** which obeys the LIFO order.
  - The ***stack_regs[id]*** is updated by changing its head to a record whose ***pc*** field points to the action done once the call has finished.
  - On top of ***stack_regs[id]*** is pushed a record with ***pc*** pointing to the first statement of the called function, and ***fp*** to the new stack frame.

```
int max(int i,int j)
{ if (i => j)
    return i;
  else return j;
}

void f1(){
   x = ...
   y = ...
   int m = max(x,y);
   …}
```

Inter-procedural Control Flow

Function call

- A function call is translated in two actions :
  - The stack frame is pushed onto the ***stack_data[id]*** which obeys the LIFO order.
  - The ***stack_regs[id]*** is updated by changing its head to a record whose ***pc*** field points to the action done once the call has finished.
  - On top of ***stack_regs[id]*** is pushed a record with ***pc*** pointing to the first statement of the called function, and ***fp*** to the new stack frame.

```
int max(int i,int j)
{ if (i => j)
    return i;
  else return j;
}

void f1(){
   x = ...
   y = ...
   int m = max(x,y);
   …}
```

$$
\begin{array}{l}
\lor \land Head(stack\_regs[id]).pc = \langle \text{“f1”}, \text{“lbl\_9”} \rangle \\
\quad \land stack\_data' = [stack\_data \text{ EXCEPT } ![id] = stack\_data[id] \circ \\
\qquad\qquad \langle load(id, Addr\_x), load(id, Addr\_y), [val \mapsto Undef] \rangle] \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \\
\qquad\qquad \langle [pc \mapsto \langle \text{“max”}, \text{“lbl\_1”} \rangle, fp \mapsto Len(stack\_data[id]) + 1] \rangle \\
\qquad\qquad \circ \langle [pc \mapsto \langle \text{“f1”}, \text{“lbl\_9.1”} \rangle, fp \mapsto Head(stack\_regs[id]).fp] \rangle \\
\qquad\qquad \circ\ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, ret \rangle
\end{array}
$$

Inter-procedural Control Flow

Function call

- A function call is translated in two actions :
  - The stack frame is pushed onto the ***stack_data[id]*** which obeys the LIFO order.
  - The ***stack_regs[id]*** is updated by changing its head to a record whose ***pc*** field points to the action done once the call has finished.
  - On top of ***stack_regs[id]*** is pushed a record with ***pc*** pointing to the first statement of the called function, and ***fp*** to the new stack frame.
  - The second action copies the return value ***ret[id]*** in the considered variable .

```
   int max(int i,int j)
1  { if (i => j)
2      return i;
3    else return j;
4  }
5
6  void f1(){
7     x = ...
8     y = ...
9     int m = max(x,y);
10    …}
```

$$
\begin{array}{l}
\lor \land Head(stack\_regs[id]).pc = \langle \text{“f1”}, \text{“lbl\_9”} \rangle \\
\quad \land stack\_data' = [stack\_data \text{ EXCEPT } ![id] = stack\_data[id] \circ \\
\qquad\qquad \langle load(id, Addr\_x), load(id, Addr\_y), [val \mapsto Undef] \rangle] \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \\
\qquad\qquad \langle [pc \mapsto \langle \text{“max”}, \text{“lbl\_1”} \rangle, fp \mapsto Len(stack\_data[id]) + 1] \rangle \\
\qquad\qquad \circ \langle [pc \mapsto \langle \text{“f1”}, \text{“lbl\_9.1”} \rangle, fp \mapsto Head(stack\_regs[id]).fp] \rangle \\
\qquad\qquad \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle mem, ret \rangle \\
\lor \land Head(stack\_regs[id]).pc = \langle \text{“f1”}, \text{“lbl\_9.1”} \rangle \\
\quad \land store(id, [loc \mapsto Addr\_f1\_m.loc, offs \mapsto Addr\_f1\_m.offs], ret[id]) \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \langle [pc \mapsto \langle \text{“f1”}, \text{“lbl\_10”} \rangle, \\
\qquad\qquad fp \mapsto Head(stack\_regs[id]).fp] \rangle \circ Tail(stack\_regs[id])] \\
\quad \land \text{UNCHANGED } \langle ret \rangle
\end{array}
$$

…

- Once the function returns:
  - The top of the ***stack_regs[id]*** is popped,
  - Its stack frame is removed from stack data[id] (using the ***SubSeq*** operator).
  - The returned value is stored on ***ret[id]***.

- Example:

```
...
int process0(){
   int i = 1;
   x = x + i;
   return x;
}
```

$$
\begin{aligned}
\vee\ \wedge\ & Head(stack\_regs[id]).pc = \langle \text{"process0"}, \text{"lbl\_3"} \rangle \\
\wedge\ & stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = Tail(stack\_regs[id])] \\
\wedge\ & stack\_data' = [stack\_data \text{ EXCEPT } ![id] = \\
& \qquad SubSeq(stack\_data[id], 1, Head(stack\_regs[id]).fp - 1)] \\
\wedge\ & ret' = [ret \text{ EXCEPT } ![id] = load(id, Addr\_x)] \\
\wedge\ & \text{UNCHANGED } \langle mem \rangle
\end{aligned}
$$

Generating the specification

- *Init* predicate that initializes all variables of the system.
  - The number of process and the entry point (initial function) of each one are specified by user. This will initialize the *stack_regs* variable.
  - The *mem* variable is initialized according to the initializers of global variables.
  - The *stack_data* is initially empty and the *ret* variable contains *Undef* value, for all processes.
- The predicate *process(id),* that defines the next-state action of the process *id*
  - It asserts that one of the function is being executed while *stack_reg[id]* is not empty.

$$process(id) \triangleq \land\ stack\_regs[id] \neq \langle\rangle$$
$$\land\ (\lor max(id) \lor process0(id) \lor process1(id))$$

```
int max(int ,int y)
{…}
void process0()
{…}
void process1()
{…}
```

- The tuple of all variables

$$vars \triangleq \langle mem, stack\_regs, stack\_data, ret\rangle$$

Generating the specification

- The next-state action ***Next*** of all processes
  - One of the process that has not finished is nondeterministically chosen to execute one step until ***stack_regs*** becomes empty.

$$Next \triangleq \lor \exists\, id \in Procs : process(id) \\ \qquad\quad \lor(\forall\, id \in Procs : (stack\_regs[id] = \langle\rangle) \land (\text{UNCHANGED } vars))$$

- The complete specification

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

Examples of properties

```
int x = 0;
int lock_var = 0; // lock global
    variable
void process0(int i){
    acquire_mutex();
    x++;
    x =  x + i;
    release_mutex();
    ...}

void process1(int j){
    acquire_mutex();
    j = x;
    x =  x + i;
    release_mutex();
    ...}
```

Critical section sc1 (lines `x++;` and `x =  x + i;` of process0)

Critical section sc2 (lines `j = x;` and `x =  x + i;` of process1)

Examples of properties

```
1 - int x = 0;
2 - int lock_var = 0; // lock global
3 -     variable
4 - void process0(int i){
5 -     acquire_mutex();
6 -     x++;
7 -     x =  x + i;
8 -     release_mutex();
9 -     ...}
10 -
11 - void process1(int j){
12 -     acquire_mutex();
13 -     j = x;
14 -     x =  x + i;
15 -     release_mutex();
16 -     ...}
```

Critical section sc1 (lines 6–7)

Critical section sc2 (lines 13–14)

- Mutual exclusion

$$\forall sc1 \in \{\langle \text{“process0”}, \text{“lbl\_6”}\rangle, \langle \text{“process0”}, \text{“lbl\_7”}\rangle\} :$$
$$\forall sc2 \in \{\langle \text{“process1”}, \text{“lbl\_13”}\rangle, \langle \text{“process1”}, \text{“lbl\_14”}\rangle\} :$$
$$((Head(stack\_regs[\text{“process0”}]).pc = sc1) \Rightarrow$$
$$(Head(stack\_regs[\text{“process1”}]).pc \neq sc2))$$

Examples of properties

```
1 -int x = 0;
2 -int lock_var = 0; // lock global
3 -    variable
4 -void process0(int i){
5 -    acquire_mutex();
6 -    x++;
7 -    x =  x + i;
8 -    release_mutex();
9 -    ...}
10 -
11 -void process1(int j){
12 -    acquire_mutex();
13 -    j = x;
14 -    x =  x + i;
15 -    release_mutex();
16 -    ...}
```

Critical section sc1 (lines 6–7)

Critical section sc2 (lines 13–14)

- Mutual exclusion

$$\forall sc1 \in \{\langle \text{“process0”}, \text{“lbl\_6”}\rangle, \langle \text{“process0”}, \text{“lbl\_7”}\rangle\} : \forall sc2 \in \{\langle \text{“process1”}, \text{“lbl\_13”}\rangle, \langle \text{“process1”}, \text{“lbl\_14”}\rangle\} : ((Head(stack\_regs[\text{“process0”}]).pc = sc1) \Rightarrow (Head(stack\_regs[\text{“process1”}]).pc \neq sc2))$$

- Termination

$$\Diamond(\forall\ id \in \{\text{“process0”}, \text{“process1”}\} : Head(stack\_regs[id]).pc = \langle\rangle)$$

Considering fairness assumptions in the specification

Conclusion

Conclusion

- C2TLA+: A translator from C to TLA+ specification that can be checked by TLC.
- The translation is based on a set of translation rules.

Future works

- Handle missing features.
- Catch all C runtime-errors.
- Using TLAPS to prove that a (translated) specification implements an abstract one.

FROM RESEARCH TO INDUSTRY

cea tech

C2TLA+: A translator from C to TLA+

Amira METHNI (CEA/CNAM)
Matthieu LEMERRE (CEA) & Belgacem BEN HEDIA (CEA)
Serge HADDAD (ENS Cachan) & Kamel BARKAOUI (CNAM)

www.cea.fr

leti & list

June 3, 2014

Thank you

Questions ??

Centre de Grenoble
17 rue des Martyrs
38054 Grenoble Cedex

Centre de Saclay
Nano-Innov PC 172
91191 Gif sur Yvette Cedex

FROM RESEARCH TO INDUSTRY

cea tech

C2TLA+: A translator from C to TLA+

Amira METHNI (CEA/CNAM)
Matthieu LEMERRE (CEA) & Belgacem BEN HEDIA (CEA)
Serge HADDAD (ENS Cachan) & Kamel BARKAOUI (CNAM)

www.cea.fr

June 3, 2014

leti & list