Critical Embedded Real-Time Systems

Systèmes Temps Réel Embarqués Critiques

STREC - WCET - Static Analysis

Florian Brandner
Télécom Paris

Outline

Sub-Module Outline

1. **The `While` Language**
   - The Language and its Syntax
   - "Code Generation"
   - Control-Flow Graphs
2. **Basic Data-Flow Analysis**
   - Abstract Domains
   - Transfer Functions
   - Meet/Join
   - Constant and Value Range Analysis
3. Worst-Case Execution Time Analysis
4. Static Cache Analysis

Books

Helpful books might be:

- **Data-Flow Analysis Theory and Practice**
  by Uday P. Khedker, Amitabha Sanyal and Bageshri Karkare
  - Chapter 3:
    Covers lattices and transfer Functions (today) as well as data-flow equations and MFP and MOP solutions (next time).
  - Chapter 4.2.3:
    Covers a simple analysis called Constant Propagation (today).
  - Chapter 7:
    Covers Inter-procedural analysis (next time).
  - Chapter 9:
    Covers call contexts (next time).

- **Principles of Program Analysis**
  by Flemming Nielson, Hanne Riis Nielson, and Chris Hankin
  - Very formal and thus hard to read for beginners
  - But a great reference . . .

While

Why `While`?

`While` is a toy language ...

- Types:
  - Integer
  - Pointers
  - Arrays of fixed size
- Variables:
  - Global and local variables
- Functions
  - `if–then–else`
  - `while` loops
  - Expressions with some basic operators (`=`, `+`, `-`, `*`, `/`, `==`, `!=`, `<`, `<=`, `*`, `&`, `[]`)

TELECOM ParisTech

Why `While`?

`While` is a toy language . . .

- Types:
  - Integer
  - Pointers
  - Arrays of fixed size
- Variables:
  - Global and local variables
- Functions
  - `if−then−else`
  - `while` loops
  - Expressions with some basic operators (`=`, `+`, `−`, `*`, `/`, `==`, `!=`, `<`, `<=`, `*`, `&`, `[]`)

Allows us to reason about realistic, but still simple, programs . . .

While Syntax: Programs

```
program
   :    definition* EOF
   ;

definition
   : var_def ';'
   | fun_def
   ;
```

Programs are a series of global variable or function definitions.

[0] This is the actual ANTLR4 grammar (without the code for type checking).

While Syntax: Global Variables

```
N : '-'? [0-9]+;
S : '"' ~["]* '"';

int_init : '=' N ;

array_init
   : '=' '{' N (',' N)* '}'
   | '=' S
   ;

var_def
   : 'int' ID (int_init)?
   | 'int' ID '[' N ']' (array_init)?
   | 'int' ID '[' ']' (array_init)?
   | 'int' '*' ID
   ;
```

While Example: Global Variables

```
int a;                           // Uninitialized integer variable
int b = -5;                      // Integer variable to -5

int c[6];                        // Integer array of size 6
int d[2]; = {1, 2};              // Array with initial values
int e[]; = {1, 2};               // Array of size 2 (deduced)
int f[]; = "Hello World!";       // Array initialized from string

int *g;                          // A pointer (never initialized)
```

While Syntax: Functions

```
param_decl
   : 'int' ID
   | 'int' ID '[' N ']'
   | 'int' '*' ID
   ;

parameters : '(' param_decl (',' param_decl)* ')' ;

fun_body : (statement ';')* ;

fun_def
   : 'fun' ('*')? ID parameters?
     'begin'
       fun_body
     'end'
   ;
```

While Example: Functions

```
fun max(int a, int b)
begin
  // body goes here
end

fun swap(int *a, int *b)
begin
  // body goes here
end

fun *min(int *data, int n)
begin
  // body goes here
end
```

While Syntax: Statements

```
statement
   : var_def
   | ID '=' expr
   | ID ('[' expr ']')? '=' expr
   | '*' expr '=' expr
   | expr
   | 'if' expr 'then'
     stmtsThen
     ('else' stmtsElse)? 'end'
   | 'while' expr 'do'
     stmtsWhile 'end'
   | 'return' expr
   ;
```

`While` Syntax: Expressions

```
expr : N
     | ID
     | ID '[' expr ']'
     | ID ('(' expr (',' expr)* ')')
     | '&' ID
     | '&' ID '[' expr ']'
     | '*' expr
     | expr '==' expr
     | expr '!=' expr
     | expr '<' expr
     | expr '<=' expr
     | expr '*' expr
     | expr '/' expr
     | expr '+' expr
     | expr '-' expr
     | '(' expr ')'
     ;
```

While Examples (1)

```
fun max(int a, int b)
begin
  if a < b then
    return b;
  else
    return a;
  end;
end

fun swap(int *a, int *b)
begin
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
end
```


TELECOM ParisTech

While Examples (2)

```
fun *min(int *data, int n)
begin
  int *p;
  int *e;
  p = data;
  e = data + n;
  while data < e do
    if *data < *p then
      p = data;
    end;
    data = data + 1;
  end;
  return p;
end
```

TELECOM ParisTech

`While` Builtins

Some special functions for I/O:

- `printint(int)`
- `printchar(int)`
- `printptr(int*)`
- `printstring(int*)` (null terminated)
- `exit(int)`

Code Generation

`While` Code: An Overview

Generation of simplified code:

- Functions consist of **basic blocks**
- Basic blocks consist of simple **instructions**
- Instructions:
  - An opcode
  - A list of operands

`While` Code: Opcodes and their Operands

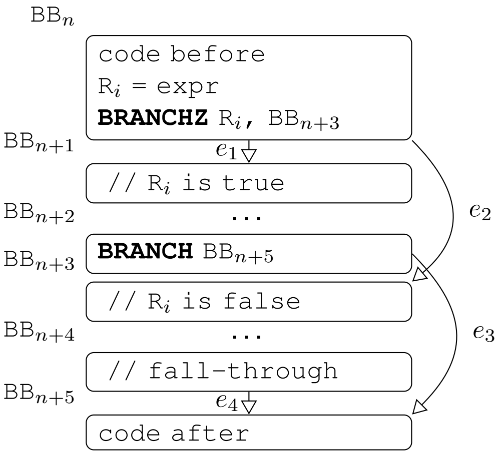
WCALL	Ops: Fun OpD = Arg1, Arg2, ... ArgN
WLOAD	Ops: OpD = [BaseAddress + Offset]
WSTORE	Ops: [BaseAddress + Offset] = ValueToStore
WPLUS	Ops: OpD = OpA + OpB
WMINUS	Ops: OpD = OpA - OpB
WMULT	Ops: OpD = OpA * OpB
WDIV	Ops: OpD = OpA / OpB
WEQUAL	Ops: OpD = OpA == OpB
WUNEQUAL	Ops: OpD = OpA != OpB
WLESS	Ops: OpD = OpA < OpB
WLESSEQUAL	Ops: OpD = OpA <= OpB
WBRANCHZ	Ops: Cond, BB
WBRANCH	Ops: BB
WRETURN	Ops: ValueToReturn

`While` Code: Instruction Operands

- Constants:
  A constant integer number
- Symbolic Register (`Ri`):
  Numbered, starting from 0 for each function
- Frame Pointer (`FP`):
  A sort of special register
- Basic Block (`BBi`):
  Index of a basic block of the current function
- Function:
  Index of a function or builtin

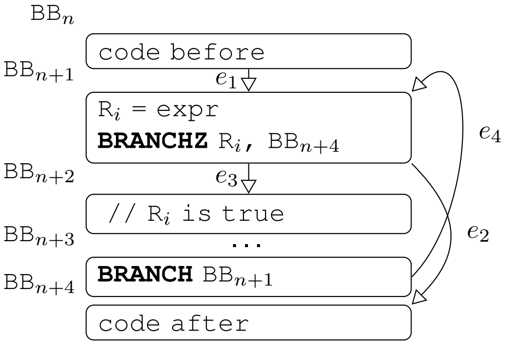
While Code: if–then–else

```
// code before
if expr then
begin
   // code when true
end
else
begin
   // code when false
end;
// code after
```



While Code: `while`

```
// code before
while expr
begin
  // code when true
end
// code after
```



While Code: Example

```
fun 0: min: [main::BB3::1]
  # 2
  # a: FP + 0 R0   {}
  # b: FP + 1 R1   {}
BB0: [] -> [BB1 (FT), BB2 (BT)]
   0:       WLOAD   R0 , FP , 0          # 21:0: a
   1:       WLOAD   R1 , FP , 1          # 21:0: b
   2:       WLESS   R2 , R0 , R1         # 23:5: a b
   3:    WBRANCHZ   R2 , BB2             # 23:2
BB1: [BB0] -> []
   0:     WRETURN   R0                   # 24:4: a
BB2: [BB0] -> [BB3 (FT)]
   0:     WRETURN   R1                   # 26:4: b
BB3: [BB2] -> []
   0:     WRETURN   0                    # 28:0
```

The function's numeric ID.

TELECOM ParisTech

While Code: Example

```
fun 0: min: [main::BB3::1]
  # 2
  # a: FP + 0 R0  {}
  # b: FP + 1 R1  {}
BB0: [] -> [BB1 (FT), BB2 (BT)]
   0:       WLOAD  R0 , FP , 0          # 21:0: a
   1:       WLOAD  R1 , FP , 1          # 21:0: b
   2:       WLESS  R2 , R0 , R1         # 23:5: a b
   3:   WBRANCHZ  R2 , BB2              # 23:2
BB1: [BB0] -> []
   0:     WRETURN  R0                   # 24:4: a
BB2: [BB0] -> [BB3 (FT)]
   0:     WRETURN  R1                   # 26:4: b
BB3: [BB2] -> []
   0:     WRETURN  0                    # 28:0
```

The function’s name.

While Code: Example

```
fun 0: min: [main::BB3::1]
  # 2
  # a: FP + 0 R0  {}
  # b: FP + 1 R1  {}
BB0: [] -> [BB1 (FT), BB2 (BT)]
   0:       WLOAD  R0 , FP , 0          # 21:0: a
   1:       WLOAD  R1 , FP , 1          # 21:0: b
   2:       WLESS  R2 , R0 , R1         # 23:5: a b
   3:    WBRANCHZ  R2 , BB2             # 23:2
BB1: [BB0] -> []
   0:     WRETURN  R0                   # 24:4: a
BB2: [BB0] -> [BB3 (FT)]
   0:     WRETURN  R1                   # 26:4: b
BB3: [BB2] -> []
   0:     WRETURN  0                    # 28:0
```

`WCALL` instructions that refer to the function.

While Code: Example

```
fun 0: min: [main::BB3::1]
  # 2
  # a: FP + 0 R0   {}
  # b: FP + 1 R1   {}
BB0: [] -> [BB1 (FT), BB2 (BT)]
   0:       WLOAD   R0 , FP , 0          # 21:0: a
   1:       WLOAD   R1 , FP , 1          # 21:0: b
   2:       WLESS   R2 , R0 , R1         # 23:5: a b
   3:    WBRANCHZ   R2 , BB2             # 23:2
BB1: [BB0] -> []
   0:     WRETURN   R0                   # 24:4: a
BB2: [BB0] -> [BB3 (FT)]
   0:     WRETURN   R1                   # 26:4: b
BB3: [BB2] -> []
   0:     WRETURN   0                    # 28:0
```

The number of local variables, i.e., the frame size.

`While` Code: Example

```
fun 0: min: [main::BB3::1]
  # 2
  # a: FP + 0 R0  {}
  # b: FP + 1 R1  {}
BB0: [] -> [BB1 (FT), BB2 (BT)]
   0:       WLOAD   R0 , FP , 0          # 21:0: a
   1:       WLOAD   R1 , FP , 1          # 21:0: b
   2:       WLESS   R2 , R0 , R1         # 23:5: a b
   3:    WBRANCHZ   R2 , BB2             # 23:2
BB1: [BB0] -> []
   0:     WRETURN   R0                   # 24:4: a
BB2: [BB0] -> [BB3 (FT)]
   0:     WRETURN   R1                   # 26:4: b
BB3: [BB2] -> []
   0:     WRETURN   0                    # 28:0
```

The address of a variable within the frame.

While Code: Example

```
fun 0: min: [main::BB3::1]
  # 2
  # a: FP + 0 R0  {}
  # b: FP + 1 R1  {}
BB0: [] -> [BB1 (FT), BB2 (BT)]
   0:       WLOAD  R0 , FP , 0          # 21:0: a
   1:       WLOAD  R1 , FP , 1          # 21:0: b
   2:       WLESS  R2 , R0 , R1         # 23:5: a b
   3:   WBRANCHZ   R2 , BB2             # 23:2
BB1: [BB0] -> []
   0:     WRETURN  R0                   # 24:4: a
BB2: [BB0] -> [BB3 (FT)]
   0:     WRETURN  R1                   # 26:4: b
BB3: [BB2] -> []
   0:     WRETURN  0                    # 28:0
```

The register of a variable – only when address is not taken.

TELECOM ParisTech

While Code: Example

```
fun 0: min: [main::BB3::1]
  # 2
  # a: FP + 0 R0   {}
  # b: FP + 1 R1   {}
BB0: [] -> [BB1 (FT), BB2 (BT)]
   0:       WLOAD   R0 , FP , 0          # 21:0: a
   1:       WLOAD   R1 , FP , 1          # 21:0: b
   2:       WLESS   R2 , R0 , R1         # 23:5: a b
   3:    WBRANCHZ   R2 , BB2             # 23:2
BB1: [BB0] -> []
   0:     WRETURN   R0                   # 24:4: a
BB2: [BB0] -> [BB3 (FT)]
   0:     WRETURN   R1                   # 26:4: b
BB3: [BB2] -> []
   0:     WRETURN   0                    # 28:0
```

The predecessors of a basic block.

While Code: Example

```
fun 0: min: [main::BB3::1]
  # 2
  # a: FP + 0 R0  {}
  # b: FP + 1 R1  {}
BB0: [] -> [BB1 (FT)], BB2 (BT)]
   0:     WLOAD   R0 , FP , 0          # 21:0: a
   1:     WLOAD   R1 , FP , 1          # 21:0: b
   2:     WLESS   R2 , R0 , R1         # 23:5: a b
   3:  WBRANCHZ   R2 , BB2             # 23:2
BB1: [BB0] -> []
   0:   WRETURN   R0                   # 24:4: a
BB2: [BB0] -> [BB3 (FT)]
   0:   WRETURN   R1                   # 26:4: b
BB3: [BB2] -> []
   0:   WRETURN   0                    # 28:0
```

The fall-through successor of a basic block.

While Code: Example

```
fun 0: min: [main::BB3::1]
  # 2
  # a: FP + 0 R0   {}
  # b: FP + 1 R1   {}
BB0: [] -> [BB1 (FT), BB2 (BT)]
   0:        WLOAD  R0 , FP , 0         # 21:0: a
   1:        WLOAD  R1 , FP , 1         # 21:0: b
   2:        WLESS  R2 , R0 , R1        # 23:5: a b
   3:     WBRANCHZ  R2 , BB2            # 23:2
BB1: [BB0] -> []
   0:      WRETURN  R0                  # 24:4: a
BB2: [BB0] -> [BB3 (FT)]
   0:      WRETURN  R1                  # 26:4: b
BB3: [BB2] -> []
   0:      WRETURN  0                   # 28:0
```

The successor of a basic block when the branch is taken.

`While` Code: Example

```
fun 0: min: [main::BB3::1]
  # 2
  # a: FP + 0 R0   {}
  # b: FP + 1 R1   {}
BB0: [] -> [BB1 (FT), BB2 (BT)]
    0:       WLOAD   R0 , FP , 0          # 21:0: a
    1:       WLOAD   R1 , FP , 1          # 21:0: b
    2:       WLESS   R2 , R0 , R1         # 23:5: a b
    3:    WBRANCHZ   R2 , BB2             # 23:2
BB1: [BB0] -> []
    0:     WRETURN   R0                   # 24:4: a
BB2: [BB0] -> [BB3 (FT)]
    0:     WRETURN   R1                   # 26:4: b
BB3: [BB2] -> []
    0:     WRETURN   0                    # 28:0
```

Debug information (line number, and variables accessed).

Control-Flow Graphs

Control-Flow Graph

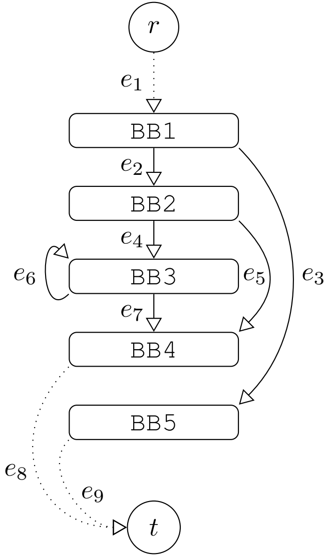
Data structure to represent code:

- Represented as a form of graph
- Graph nodes:
  - Individual instructions or
  - Sequences of instructions (aka. **basic block**)
- Graph edges:
  - Link from a graph node (instruction) to another
  - Instructions that might execute after executing an instruction
    (Basic blocks that might execute after executing a basic block)
- This allows to represent all possible executions of a program from start to end

Example 1: Control-Flow Graph (in While)

```
fun 0: min: [main::BB3::1]
  # 2
  # a: FP + 0 R0   {}
  # b: FP + 1 R1   {}
BB0: [] -> [BB1 (FT), BB2 (BT)]
   0:        WLOAD   R0 , FP , 0        # 21:0: a
   1:        WLOAD   R1 , FP , 1        # 21:0: b
   2:        WLESS   R2 , R0 , R1       # 23:5: a b
   3:     WBRANCHZ   R2 , BB2           # 23:2
BB1: [BB0] -> []
   0:      WRETURN   R0                 # 24:4: a
BB2: [BB0] -> [BB3 (FT)]
   0:      WRETURN   R1                 # 26:4: b
BB3: [BB2] -> []
   0:      WRETURN   0                  # 28:0
```

Example 2: Control-Flow Graph (abstract)



Program Semantics

Control-flow graphs are merely a program representation:

- A CFG only indicates which instructions may succeed/proceed other instructions (or basic blocks)
- A CFG does not say anything about program semantics (What is the program doing?)
- The semantics depends on the instructions within the CFG

Program Semantics

Control-flow graphs are merely a program representation:

- A CFG only indicates which instructions may succeed/proceed other instructions (or basic blocks)
- A CFG does not say anything about program semantics (What is the program doing?)
- The semantics depends on the instructions within the CFG

We need something in addition to reason about programs . . .

Basic Data-Flow Analysis

aka. Abstract Interpretation

Data-Flow Analysis

One technique to *reason* about programs:

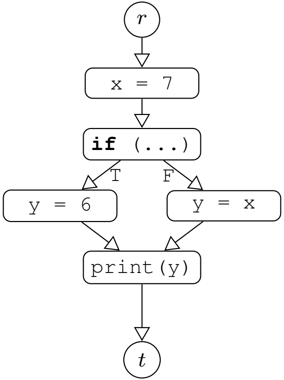
- This is often called **static analysis**
- Model the flow of information through a program
- Based on a generic *framework*
  - Abstractions (aka. Domain)
  - Transformation functions (Domain $\rightarrow$ Domain)
  - Meet/join operator (Domain $\times$ Domain $\rightarrow$ Domain)
- Given an instance of a framework
  - Build and solve data-flow equations
  - Obtain over- or under-approximation of program behavior

Example: Constant Propagation

Determine whether a variable always has a constant value:

```
x = 7;
if (...)
  y = 6;
else
  y = x;
print(y);
```

(a) Program source

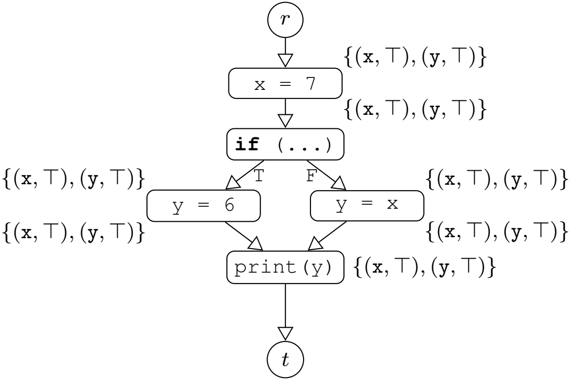


(b) Machine-level control-flow graph

Example: Constant Propagation

Associate each instruction with information on variable values:

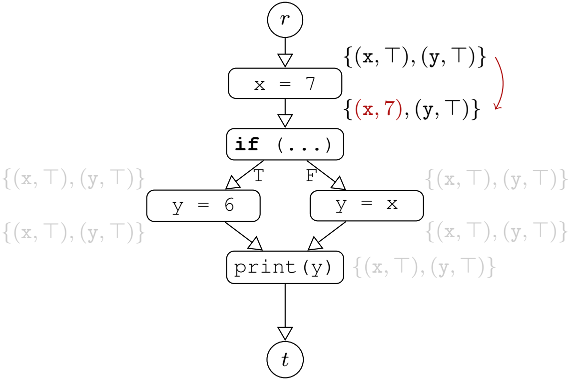
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

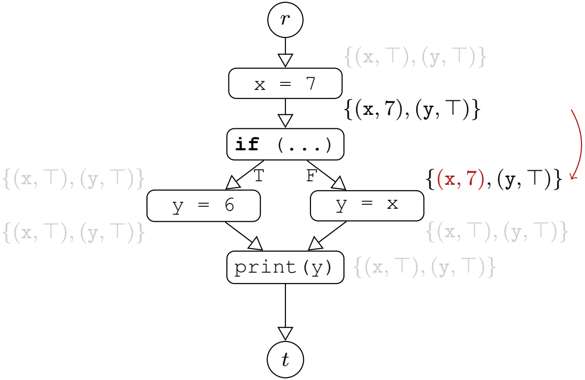
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

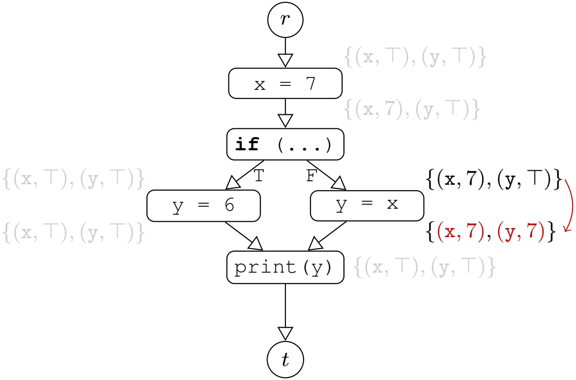
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

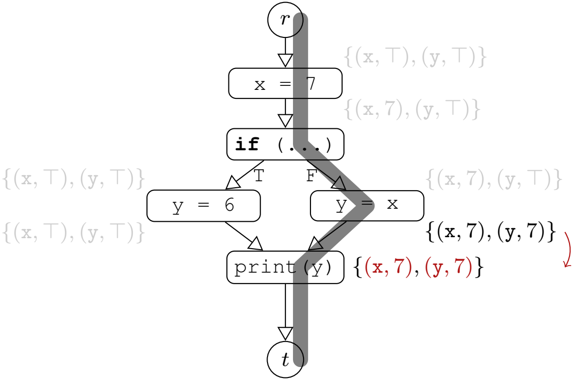
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

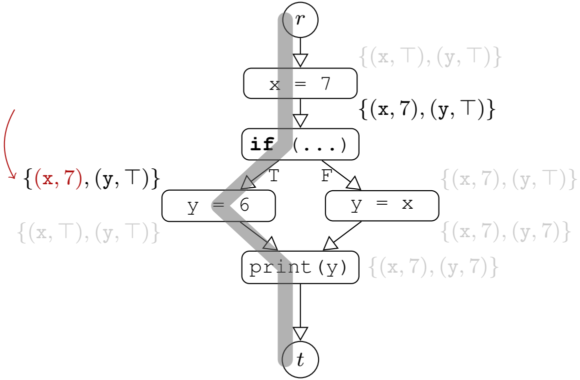
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

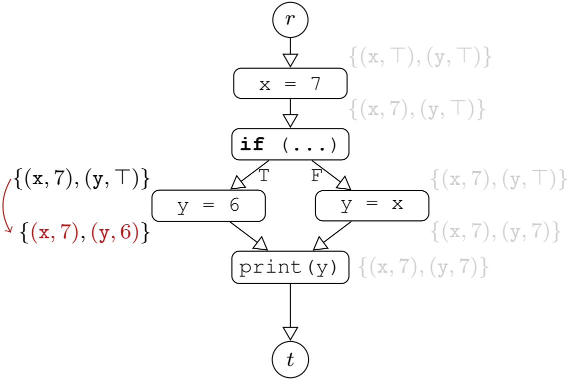
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

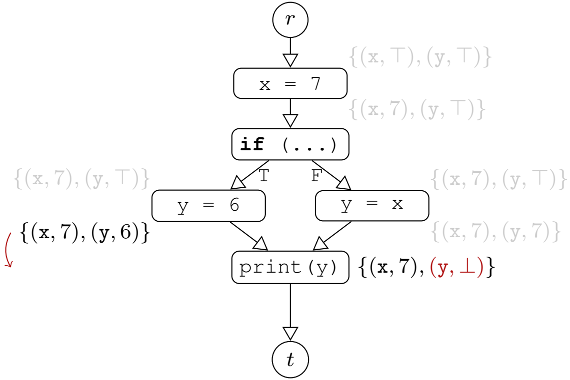
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

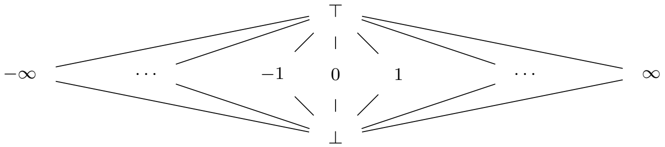
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Abstract Domain

Represents information known about the program:

- Based on partial orders (lattices)
- Information is refined by descending the lattice
- Special elements:
  - $\top$ (Top): The top-most element in the lattice, representing that *no* information is yet available
  - $\bot$ (Bottom): The least element, representing contradicting information
- Example: constant propagation



Transfer Functions

Transform the information Domain $\to$ Domain

- Capture the effect of instructions on the analysis information
- Can be almost freely defined

- Example: constant propagation

$$t(i, I) = \begin{cases} I \setminus \{(v,x) | (v,x) \in I\} \cup \{(v, \hat{c})\} & \text{, if } i \text{ is } \mathtt{v} = \hat{c} \\ I \setminus \{(v,x) | (v,x) \in I\} \cup \{(v,x) | (w,x) \in I\} & \text{, if } i \text{ is } \mathtt{v} = \mathtt{w} \\ I \setminus \{(v,x) | (v,x) \in I\} \cup \{(v, \perp)\} & \text{, if } i \text{ is } \mathtt{v} = \ldots \\ I & \text{, otherwise.} \end{cases}$$

Meet/Join Operation

Combine information at control-flow joins:

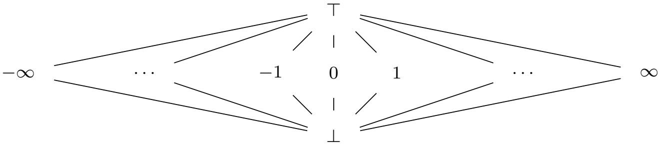
- Find least upper/greatest lower bound of two values
- Need to satisfy certain properties
  - Monotonicity ensures termination
  - Distributivity ensures optimal solution using iterative solving

- Notation:
  - $a \sqcap b$ (meet operator):
    smallest common ancestor of *a* and *b*
  - $a \sqcup b$ (join operator):
    greatest common descendent of *a* and *b*

Example: Join of Constant Propagation

The lattice for constant propagation is shown below:

- $1 \sqcup 2 = \bot$:
  The variable is either 1 or 2 depending on the predecessor. After a join we know that it is not constant, i.e., $\bot$.
- $\top \sqcup 2 = 2$:
  The variable is 2 at one predecessor. No information is available for the other predecessor. After a join the variable could still be constant, i.e., 2.



Value Range Analysis

Value Range Analysis

Determine for each variable the range of possible values:

- Extension of constant propagation (from before)
- Find constant lower- and upper-bounds for each variable
- We will only consider a simplified analysis here

- What is done with it?
  - Needed for cache analysis (access addresses)
  - Used in loop bounds analysis (loop bounds)
  - Used to detect infeasible conditions (flow-facts)

Value Range Analysis in a Nutshell

Domain:

- Set of triples over all program variables
- Variable $\times \mathbb{N} \times \mathbb{N}$

Transfer functions:

- Perform arithmetic on value ranges (interval arithmetic)
- Example: Addition
  $[a, b] + [c, d] = [a + c, b + d]$

Join operator:

- $[a, b] \sqcup [c, d] = [\min(a, c), \max(b, d)]$

TELECOM ParisTech

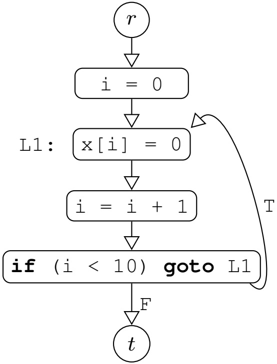
Group Exercise: Range Analysis

Determine the range of memory addresses accessed by `x[i]`:

- Assume that `x` is a global variable at address `0x100`
- Each element of `x` is 4 bytes large
- What are the initial states of the analysis?
- Which role plays the condition `if (i <10)`?

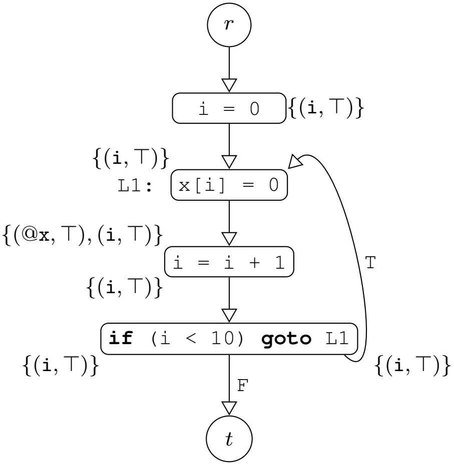
```
for(i = 0; i < 10; i++)
  x[i] = 0;
```

(a) Program source

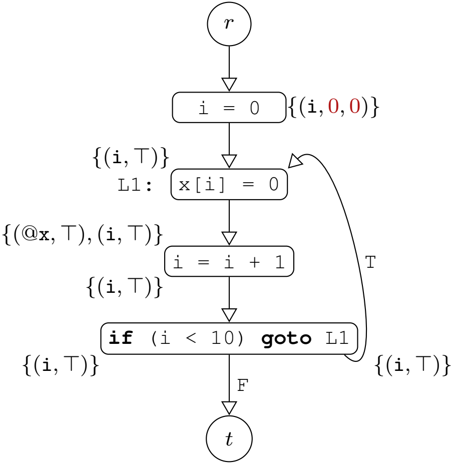


(b) Machine-level control-flow graph

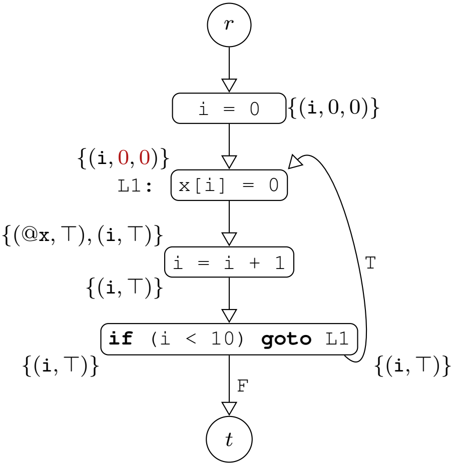
Example: Range Analysis



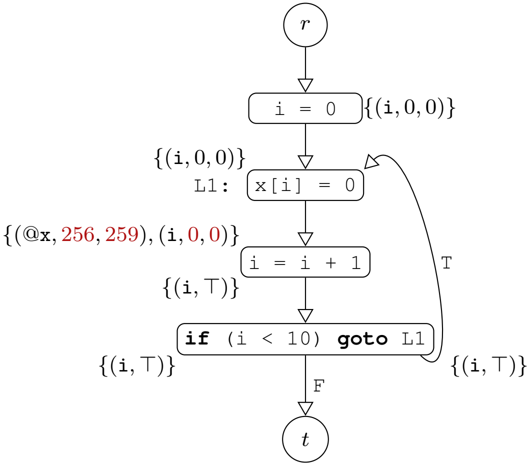
Example: Range Analysis



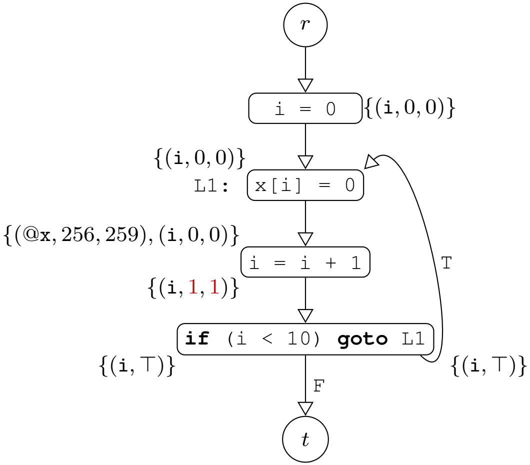
Example: Range Analysis



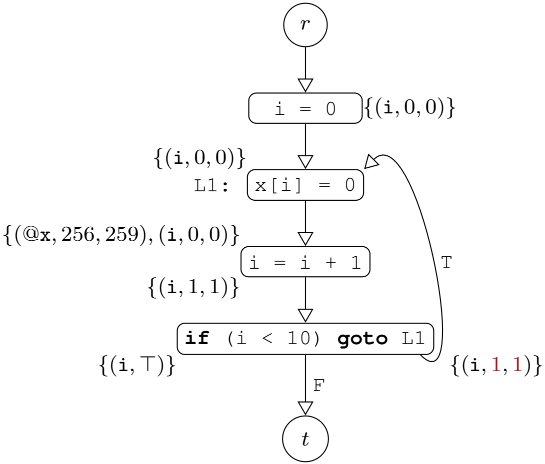
Example: Range Analysis



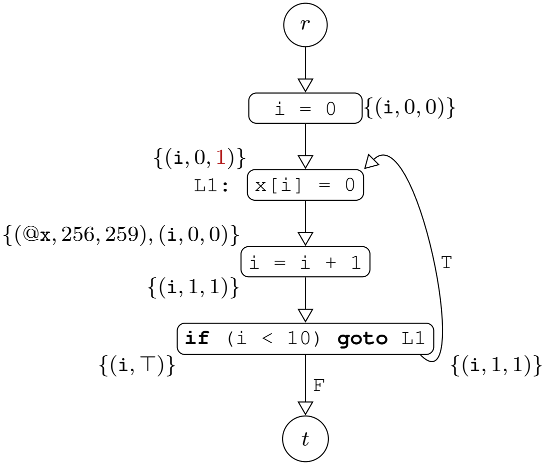
Example: Range Analysis



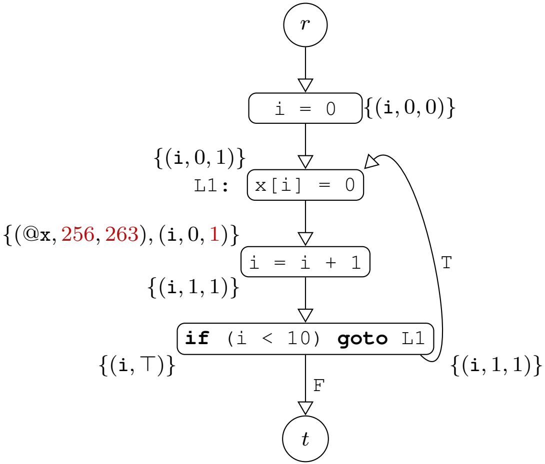
Example: Range Analysis



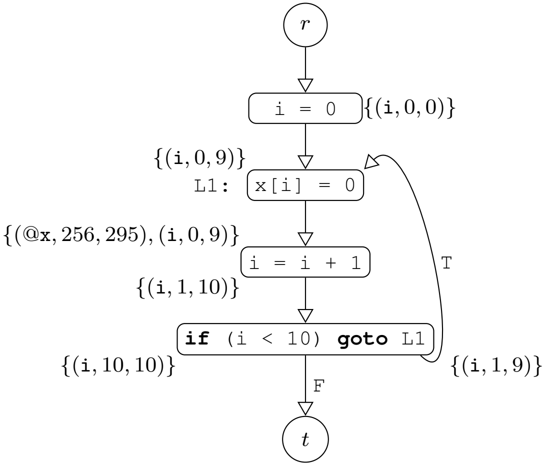
Example: Range Analysis



Example: Range Analysis



Example: Range Analysis



Summary

- `While` Language
  - Syntax and basic types
  - “Code generation”
  - Control-Flow Graphs
- Data-Flow Analysis
  - Abstract Domain (lattices)
  - Transfer Functions
  - Meet/Join Operators
- In the Lab:
  - Work with `While` (download, compile, run, . . .)
  - Study existing (partial) analysis
  - Complete existing (partial) analysis