The imagination driving Australia's ICT future.

Static Analysis

Ralf Huuck

Australian Government
Department of Communications, Information Technology and the Arts
Australian Research Council

The imagination driving Australia's ICT future.

Outline

- Introduction
- Basic Definitions
- Data Flow Analysis
- Abstract Interpretation
- Syntactical Model Checking
- Summary

The imagination driving Australia's ICT future.

Introduction

The imagination driving Australia's ICT future.

What is Static Analysis?

Static Analysis subsumes all methods which derive information from programs without actually executing them.

Static analysis is the term applied to the analysis of computer software that is performed without actually executing programs.

- wikipedia

The Trouble with Software

The state space of programs is in theory infinite.
Computation depends on

- integers
- reals
- etc.

In reality it is finite, e.g., 32-bit representations, which is still practically infinite when exploring all combinations.

The imagination driving Australia's ICT future.

Decidability

Properties on infinite state spaces are typically **undecidable**, i.e., there is no general algorithm to decide if they are true or false.

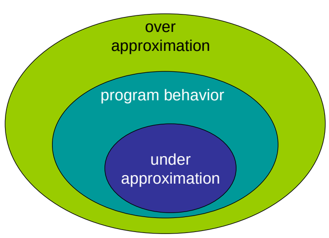
Alan Turing
1912-1954

Rice's theorem: Any nontrivial property about the language recognized by a Turing machine is undecidable.

But we can still attempt to give useful **approximate** solutions.

The imagination driving Australia's ICT future.

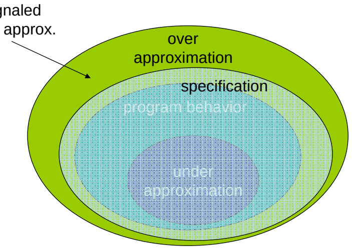
Approximations



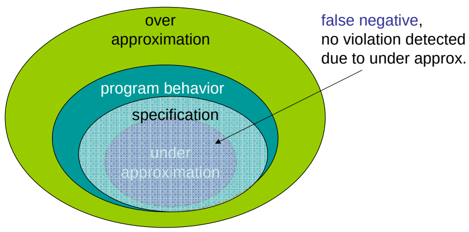
The imagination driving Australia's ICT future.

Approximations

false alarm,
violation signaled
due to over approx.



Approximations



The imagination driving Australia's ICT future. NATIONAL ICT AUSTRALIA

Definitions

The imagination driving Australia's ICT future. NATIONAL ICT AUSTRALIA

Lattices

partial order: $L=(S,\sqsubseteq)$, set S, binary relation $\sqsubseteq$ s.t.

- reflexive: $\forall\, x\in S : x \sqsubseteq x$
- transitive: $\forall\, x,y,z\in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
- anti-symmetric: $\forall\, x,y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x=y$

y upper bound of $X\subseteq S$ ($X\sqsubseteq y$): $y\in S$ and $\forall\, x\in X: x \sqsubseteq y$
least upper bound $\sqcup X$: $X\sqsubseteq \sqcup X$ and $\forall\, y\in S: X\sqsubseteq y \Rightarrow \sqcup X\sqsubseteq y$
Conversely, define lower bound and greatest lower bound.

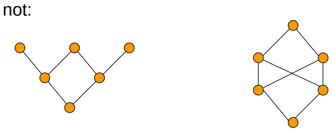
Lattice is partial order where $\sqcup X$ and $\sqcap X$ exists for all $X\subseteq S$.

The imagination driving Australia's ICT future. NATIONAL ICT AUSTRALIA

Example: Finite Lattices

Lattices:

But not:



Show: every lattice has a unique largest ($\top$) and smallest ($\perp$) element

The imagination driving Australia's ICT future.

Monotone functions

f monotone: $\forall$ x,y$\in$ S: x $\sqsubseteq$ y $\Rightarrow$ f(x) $\sqsubseteq$ f(y)

Note: compositions of monotone functions are monotone

Theorem: In lattice L with finite height every monotone function has a least fixed-point as:

$$\mathrm{fix}(f) = \bigsqcup_{i\geq 0} f^i(\perp)$$

for which f(fix(f))=fix(f).

The imagination driving Australia's ICT future.

Closure

If $L_1,\ldots,L_n$ lattices of finite height, so is the product

$L_1 \times \ldots \times L_n = \{(x_1,\ldots,x_n) \mid x_i \in L_i\}$

where $\sqsubseteq$ is defined pointwise.

The height of the product is the sum of heights of its components.

Map: finite set A, lattice L with $|A|=|L|$ then with a pointwise order

$A \mapsto L = \{[a_1 \mapsto x_1,\ldots,a_n \mapsto a_n] \mid x_i \in L\}$

is a lattice of height $|A|$*height(L).

Other compositions: +, lift, flat, are also lattices of finite height again.

The imagination driving Australia's ICT future.

Equation Systems (ES)

Equation System $F: L^n \to L^n$

$$\boxed{\begin{array}{l} x_1 = F_1(x_1,\ldots,x_n) \\ \vdots \\ x_2 = F_2(x_1,\ldots,x_n) \\ x_n = F_n(x_1,\ldots,x_n) \end{array}}$$

where

x_i variables and

$F_i : L^n \to L$ collection of monotone functions

has a least fixed point for the function

$F(x_1,\ldots,x_n)=(F_1(x_1,\ldots,x_n),\ldots,F_n(x_1,\ldots,x_n))$

The imagination driving Australia's ICT future.

Inequation System (IS)

Inequation System $F: L^n \to L^n$

$$\boxed{\begin{array}{l} x_1 \sqsubseteq F_1(x_1,\ldots,x_n) \\ \vdots \\ x_2 \sqsubseteq F_2(x_1,\ldots,x_n) \\ x_n \sqsubseteq F_n(x_1,\ldots,x_n) \end{array}}$$

We can show $x \sqsubseteq y \Leftrightarrow x = x \sqcap y$.

Thus, IS F equivalent to following ES:

$$\boxed{\begin{array}{l} x_1 = x_1 \sqcap F_1(x_1,\ldots,x_n) \\ \vdots \\ x_2 = x_2 \sqcap F_2(x_1,\ldots,x_n) \\ x_n = x_n \sqcap F_n(x_1,\ldots,x_n) \end{array}}$$

Control Flow Graph (CFG)

A control flow graph is a directed graph where nodes are program points and the edges represent the flow of control between these points.

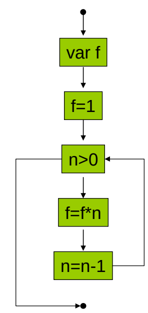
The imagination driving Australia's ICT future.

Example CFG

Program

```
fun(n) {
  var f;
  f=1;
  while (n>0) {
    f=f*n;
    n=n-1
  }
  return f;
}
```

CFG



The imagination driving Australia's ICT future.

Data Flow Analysis

The imagination driving Australia's ICT future.

Terms

Classical Data Flow Analysis is concerned about which data reaches which program point. The analysis is performed on program's CFG and expressed as an (in)equality system over a finite lattice.

Typical examples are:

- liveness
- available expressions
- very busy expressions
- reaching definitions

We will go trough each of them.

The imagination driving Australia's ICT future.

Liveness

A variable is live before a program point if it will be read in the remaining program execution without being written first.

```
fun(n) {
  var f;
  f=1;
  while (n>0) {
    f=f*n;
    n=n-1
  }
  return f;
}
```

Which variables are live at which locations? Which are not?

The imagination driving Australia's ICT future.

Liveness

Any analysis must find out the set of live variables for each location. We model the domain of live variables by the lattice $(2^{Vars},\subseteq)$.

```
fun(n) {
  var f;
  f=1;
  while (n>0) {
    f=f*n;
    n=n-1
  }
  return f;
}
```

How does the lattice $(2^{Vars},\subseteq)$ look like? Why is it a lattice?

Vars is the set of variables occurring in a program.

The imagination driving Australia's ICT future.

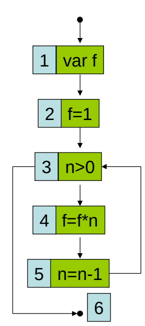
Questions

- Can we approximate (without executing the program) the set of live variables algorithmically?
- How does the set depend on our syntax?
- Can we define rules for each construct?
- How can rules lead to something that we can compute?

The imagination driving Australia's ICT future.

Question

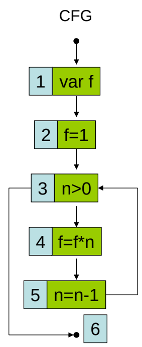
CFG



let **live(v)** denote the set of variables live before location v.

Can we define rules just depending on the current node and its predecessors or successors?

Equation System



let **live(v)** denote the set of variables live before location v.

live(1) = live(2)

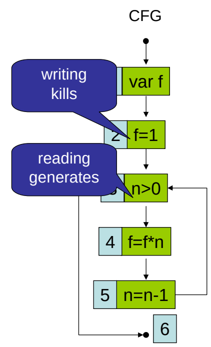
live(2) = live(3) \ {f}

live(3) = (live(4) $\cup$ live(6)) $\cup$ {n}

live(4) = (live(5) \ {f}) $\cup$ {f,n}

live(5) = (live(3) \ {n}) $\cup$ {n}
live(6) = {}

Gen/Kill



let **live(v)** denote the set of variables live before location v.

live(1) = live(2)

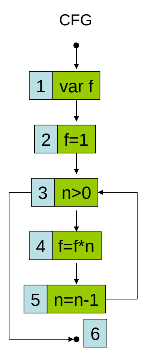
live(2) = live(3) \ {f}

live(3) = (live(4) $\cup$ live(6)) $\cup$ {n}

live(4) = (live(5) \ {f}) $\cup$ {f,n}

live(5) = (live(3) \ {n}) $\cup$ {n}
live(6) = {}

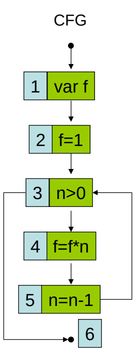
Join



let **live(v)** denote the set of variables live before location v.

use the abbreviation
join(v) = $\bigcup_{w \in succ(v)}$ **live(w)**

Join



let **live(v)** denote the set of variables live before location v.

live(1) = join(1) \ {f}

live(2) = join(2) \ {f}

live(3) = join(3) $\cup$ {n}

live(4) = (join(4) \ {f}) $\cup$ {f,n}

live(5) = (join(5) \ {n}) $\cup$ {n}
live(6) = {}

Fixed Point

The right hand-side of each equation is **monotone**, i.e., we can compute the fixed point of ES.

We are interested in the **least** fixed point.

Least fixed point: start with {} greatest start with Vars.

let **live(v)** denote the set of variables live before location v.

live(1) = join(1) \ {f}

live(2) = join(2) \ {f}

live(3) = join(3) ∪ {n}

live(4) = (join(4) \ {f}) ∪ {f,n}

live(5) = (join(5) \ {n}) ∪ {n}
live(6) = {}

Computing Fixed Point

Computation on the board.

let **live(v)** denote the set of variables live before location v.

live(1) = live(2)

live(2) = live(3) \ {f}

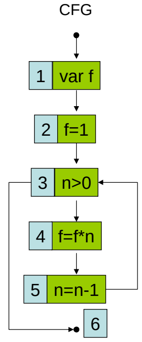
live(3) = (live(4) ∪ live(6)) ∪ {n}

live(4) = (live(5) \ {f}) ∪ {f,n}

live(5) = (live(3) \ {n}) ∪ {n}
live(6) = {}

Least Fixed Point Solution

CFG



let **live(v)** denote the set of variables live before location v.

live(1) = {n}

live(2) = {n}

live(3) = {f,n}

live(4) = {f,n}

live(5) = {f,n}
live(6) = {}

How to Compute Fixed Points

How/in which order to we iterate through the equations?

- following equation order
- randomly (requires fairness)
- worklists/priority queues
- strongly connected components (program topology)
- Gaus-Seidl elimination
- etc. etc.

There is no general best way, but some are better than others.

Intermediate Summary

Seen so far:

- data flow analysis problem can be expressed in terms of fixed point over equations
- equations depend on syntax of program points and what is coming in/going out
- many ways to compute fixed point

We have seen Join as being union of successors and computation of least fixed point. This is not always so …

Available Expressions

An expression is available after a program point if its current value has been evaluated before and none of its variables are overwritten. (Good for optimizations)

```
fun(n) {
  var f;
  f=1;
  while (n>0) {
    f=f*n;
    n=n-1
  }
  return f;
}
```

Which expressions are available at which locations? Which are not?

Available Expressions

Our analysis must find out the set of available expressions for each location. We model the domain of available expression by the lattice $(2^{Expr},\subseteq)$.

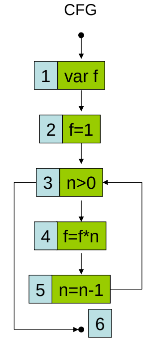
```
fun(n) {
  var f;
  f=1;
  while (n>0) {
    f=f*n;
    n=n-1
  }
  return f;
}
```

How does the lattice $(2^{Expr},\subseteq)$ look like? Why is it a lattice?

Expr is the set of expressions occurring in a program.

Equation System

CFG



let **avail(v)** denote the set of expressions available after location v.

avail(1) = {}

avail(2) = (avail(1) \ {f*n}) ∪ {1}

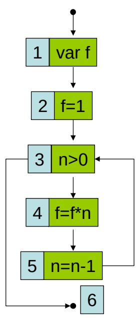
avail(3) = (avail(2) ∩ avail(5)) ∪ {n>0}

avail(4) = (avail(3) \ {f*n}) ∪ {f*n}

avail(5) = (avail(4) \ {n-1}) ∪ {n-1}
avail(6) = avail(3)

Join/Fixed Point

CFG



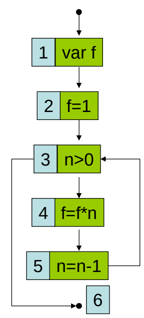
let **avail(v)** denote the set of variables available after location v.

We can use the abbreviation
join(v) $= \bigcap_{w \in pred(v)}$ **avail(w)**
to see we have again a monotone framework.

This time we compute the **greatest fixed point**, as we like to have the maximum number of available expressions.

Greatest Fixed Point Solution

CFG



let **avail(v)** denote the set of expressions available after location v.

avail(1) = {}

avail(2) = {1}

avail(3) = {n>0}

avail(4) = {n>0, f*n}

avail(5) = {f*n ,n-1}
avail(6) = {n>0}

Very Busy Expressions

An expression is very busy before a program point if it definitely will be evaluated again before its value changes.

```
fun(n) {
  var f;
  f=1;
  while (n>0) {
    f=f*n;
    n=n-1
  }
  return f;
}
```

Which expressions are very busy at which locations? Which are not?

Very Busy Expressions

Our analysis must find out the set of very busy expressions for each location. We model the domain of very busy expression by the lattice (2^{Expr},$\subseteq$).

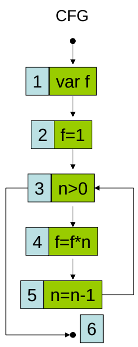
```
fun(n) {
  var f;
  f=1;
  while (n>0) {
    f=f*n;
    n=n-1
  }
  return f;
}
```

How does the lattice (2^{Expr},$\subseteq$) look like? Why is it a lattice?

Expr is the set of expressions occurring in a program.

The imagination driving Australia's ICT future.

Equation System



let **busy(v)** denote the set of expressions very busy before location v.

busy(1) = buys(2)

busy(2) = (busy(3) \ {f*n}) ∪ {1}

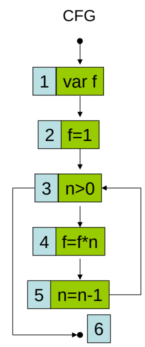
busy(3) = (busy(2) ∩ busy(5)) ∪ {n>0}

busy(4) = (busy(5) \ {f*n}) ∪ {f*n}

busy(5) = (busy(3) \ {n-1}) ∪ {n-1}
busy(6) = { }

The imagination driving Australia's ICT future.

Join/Fixed Point



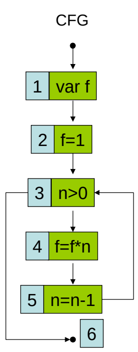
let **busy(v)** denote the set of variables very busy before location v.

We can use the abbreviation **join(v)** $= \bigcap_{w \in succ(v)}$ **busy(w)** to see we have again a monotone framework.

This time we compute the **least fixed point** again.

The imagination driving Australia's ICT future.

Least Fixed Point Solution



let **busy(v)** denote the set of expressions very busy before location v.

busy(1) = {n>0, 1}

busy(2) = {n>0, 1}

busy(3) = {n>0}

busy(4) = {n>0, n-1, f*n}

busy(5) = {n>0, n-1}
busy(6) = { }

The imagination driving Australia's ICT future.

Reaching Definition

The effect of an assignment is **reaching** after a program point if it might have defined the current values of variables.

```
fun(n) {
  var f;
  f=1;
  while (n>0) {
     f=f*n;
     n=n-1
  }
  return f;
}
```

Which assignments are reaching at which locations? Which are not?

Reaching Definitions

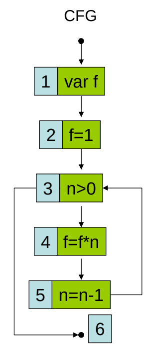
Our analysis must find out the set of reaching assignments for each location. We model the domain of reaching definitions by the lattice (2^{Assign},⊆).

```
fun(n) {
  var f;
  f=1;
  while (n>0) {
    f=f*n;
    n=n-1
  }
  return f;
}
```

How does the lattice (2^{Assign},⊆) look like? Why is it a lattice?

Assign is the set of assignments occurring in a program.

Equation System



let **reach(v)** denote the set of assignments may define variable values after location v.

reach(1) = {}

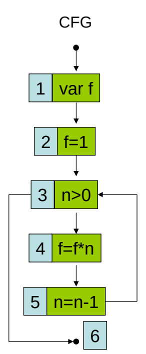
reach(2) = (reach(1) \ {f=1}) ∪ {f=1}

reach(3) = reach(2) ∪ reach(5)

reach(4) = (reach(3) \ {f=f*n,f=1}) ∪ {f=f*n}

reach(5) = (reach(4) \ {n=n-1}) ∪ {n=n-1}
reach(6) = reach(3)

Join/Fixed Point

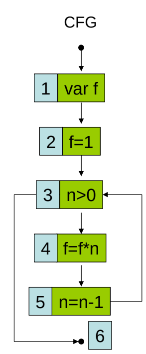


let **reach(v)** denote the set of assignments may define variable values after location v.

We can use the abbreviation **join(v)** $=\bigcup_{w\in pred(v)}$ **reach(w)** to see we have again a monotone framework.

This time we compute the **least fixed point** again.

Least Fixed Point Solution



let **reach(v)** denote the set of assignments may define variable values after location v.

reach(1) = {}

reach(2) = {f=1}

reach(3) = {f=1, f=f*n, n=n-1}

reach(4) = {f=f*n, n=n-1}

reach(5) = {f=f*n, n=n-1}
reach(6) = {f=1, f=f*n, n=n-1}

The imagination driving Australia's ICT future.

Summary Data Flow Analysis

- we have seen how to compute approximate solutions (we do not know if all the paths are executed!) to data flow problems
- Join can be intersection or union
- analysis has forward or backward nature
- depending on the problem least or greatest fixed point

The imagination driving Australia's ICT future.

Rules of Thumb

forward analysis: computes information about past behavior

backward analysis: computes information about future behavior

must analysis: information that must be true (on all paths) and computes under-approximation

may analysis: information that may be true (on at least one paths) and computes over-approximation

The imagination driving Australia's ICT future.

Abstract Interpretation

The Rough Guide

The imagination driving Australia's ICT future.

Introduction

The idea

Abstract Interpretation (AI) provides appropriate means to relate some concrete world (domain) with an abstract world.

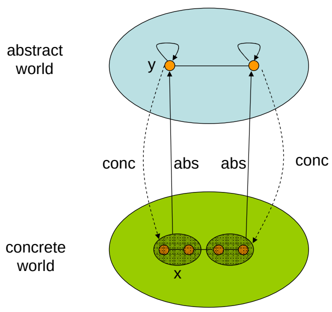
The applications and implications

Far reaching, ranging from comparing program semantics to program analysis.

Origin

First developed by Cousot&Cousot (1977/78).

Abstract Interpretation (simplified)



AI is more than abstraction as it relates abstract with concrete world and vice versa.

We require:

1. $\forall x$: abs(x) $\sqsubseteq$ y $\Leftrightarrow$ x $\sqsubseteq$ conc(y)
(Galois connection)

Operations should be safely approximated:

2. abs $\circ$ c-op $\circ$ conc $\sqsubseteq$ a-op

The imagination driving Australia's ICT future. NATIONAL ICT AUSTRALIA

Example (Interval AI)

1. Structure

Sets of integers	Intervals
{2}	[2,2]
{2,3,4}	[2,4]
{1,3,9}	[1,9]
$x \in 2^{Int}$	smallest interval comprising x

Intervals over approximate sets.

The imagination driving Australia's ICT future. NATIONAL ICT AUSTRALIA

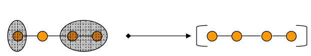
Example (Interval AI)

1. Structure

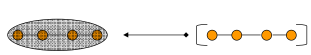
lattice (non finite) — lattice (non finite)

$(2^{Int},\subseteq)$ — (Intervals(Int),$\sqsubseteq$) smallest $\perp$ greatest $[-\infty,+\infty]$

Abstraction: $x \in 2^{Int}$ ⟷ [min x, max x]



Concretization: {lb y,…,ub y} ⟷ $y \in$ Intervals(Int)



lb = lower bound, ub = upper bound

The imagination driving Australia's ICT future. NATIONAL ICT AUSTRALIA

Example (Interval AI)

2. Operations

lattice (non finite) — lattice (non finite)

$(2^{Int},\subseteq)$ — (Intervals(Int),$\sqsubseteq$)

Concrete world: {2,5,6} + {2,3} = {4,5,7,8,9}

Abstract world: [2,6] + [2,3] = [4,9]

a) Introduce matching operator in abstract world for every operator in concrete world.

b) Check it satisfies safe approximation.

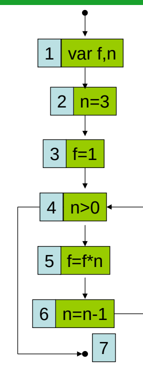
Application

We like to compute all the possible values variables can take (collecting semantics).

```
var f, n;
n=3;
f=1;
while (n>0) {
   f=f*n;
   n=n-1
}
return f;
```

The imagination driving Australia's ICT future.

Application



before each location the variables might be as follows:

2: f: {-∞, …., +∞} n: {-∞, …., +∞}

3: f: {-∞, …., +∞} n: {3}

4: f: {1} n: {3}

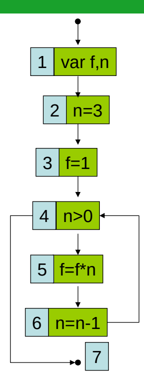
5: f: {1,3,6} n: {1,2,3}

6: f: {1,3,6} n: {0,1,2,3}

7: f: {1,3,6} n: {0}

The imagination driving Australia's ICT future.

Application



Problems:

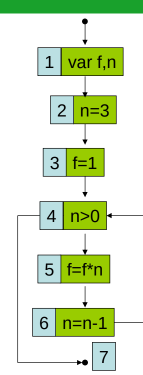
1. We might need infinite space to store values.
2. We might not be able to compute them due to non-termination.

Solution:

1. can be overcome by using interval AI (over-approximation of values)

The imagination driving Australia's ICT future.

Application



before each location the variables might be as follows:

2: f: [-∞, +∞] n: [-∞, +∞]

3: f: [-∞, +∞] n: [3,3]

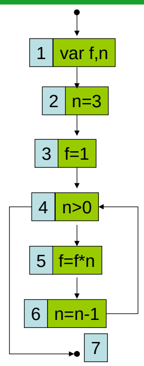
4: f: [1,1] n: [3,3]

5: f: [1,6] n: [1,3]

6: f: [1,6] n: [0,3]

7: f: [1,6] n: [0,0]

Application



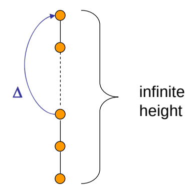
Problems:

1. We might need infinite space to store values.
2. We might not be able to compute them due to non-termination.

Solution:

1. can be overcome by using interval AI (over-approximation of all values)
2. not that easy: interval lattice has infinite width (not a problem) and infinite height (problem!).

Acceleration



AI:

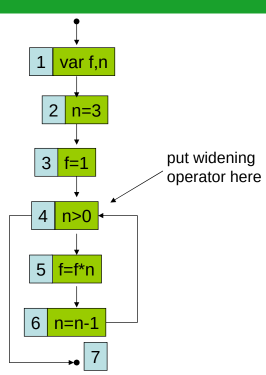
We have relation between abstract and concrete domain.

We have relation between abstract and concrete operators.

In order to deal with infinite lattices (i.e. to compute a fixed point in finite time) we introduce an extra operator that can "jump" infinitely high.

Such an acceleration operator is called **widening** operator. Sometimes people speak of dynamic approximation operator.

Application



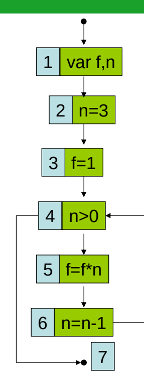
Idea:

1. if variable value increased compared to previous iteration jump to $+\infty$, if decreased to $-\infty$.

2. goto 2

Since our operations are monotone this is safe.

Application



Result:

2: f: $[-\infty, +\infty]$ n: $[-\infty, +\infty]$

3: f: $[-\infty, +\infty]$ n: $[3,3]$

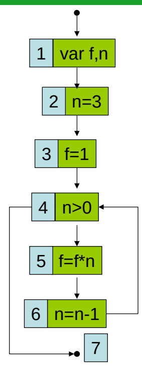
4: f: $[1,1]$ n: $[3,3]$

5: f: $[1,+\infty]$ n: $[-\infty,3]$

6: f: $[1, +\infty]$ n: $[-\infty,3]$

after first acceleration

Application



Result:

2: f: [$-\infty$, $+\infty$] n: [$-\infty$, $+\infty$]

3: f: [$-\infty$, $+\infty$] n: [3,3]

4: f: [1,1] n: [3,3]

5: f: [$-\infty$,$+\infty$] n: [$-\infty$,$+\infty$]

6: f: [$-\infty$, $+\infty$] n: [$-\infty$,$+\infty$]

6: f: [$-\infty$, $+\infty$] n: [$-\infty$,$+\infty$]

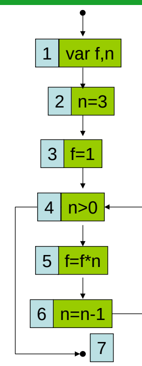
after second acceleration

approximation is very coarse

can narrow it down using condition as constraint

The imagination driving Australia's ICT future.

Application



Result:

2: f: [$-\infty$, $+\infty$] n: [$-\infty$, $+\infty$]

3: f: [$-\infty$, $+\infty$] n: [3,3]

4: f: [1,1] n: [3,3]

5: f: [$-\infty$,$+\infty$] n: [$-\infty$,$+\infty$]

6: f: [$-\infty$, $+\infty$] n: [$-\infty$,$+\infty$]

6: f: [$-\infty$, $+\infty$] n: [-\infty,0]

We might loose a lot of information, but we are still able to tell that n>0.

The imagination driving Australia's ICT future.

Summary

- AI relates abstract and concrete worlds (structure + operations)
- termination/safe approximation can be enforced by acceleration techniques

There are domains that capture more information, e.g., ployhedra.

AI is good for range approximation, i.e., array access, range check of operations, general buffer overflows.

The imagination driving Australia's ICT future.

Model Checking Syntax

Just some ideas …

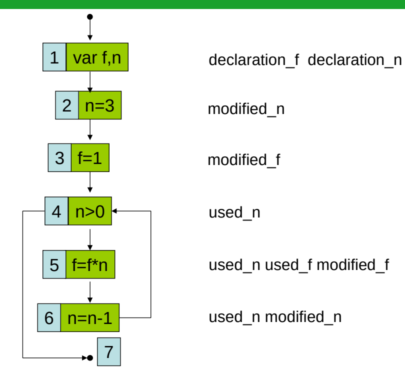
Introduction

Syntax gives us some information:

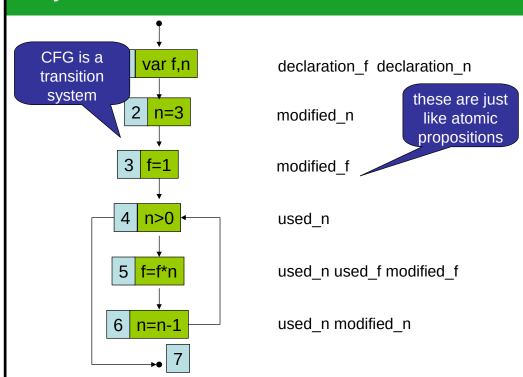
- when are variable is used
- when a variable is declared
- when a variable is modified
- etc.

Can we make use of it to find bugs in programs?

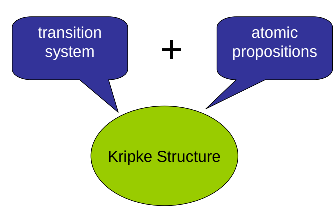
Syntactical Information



Syntactical Information

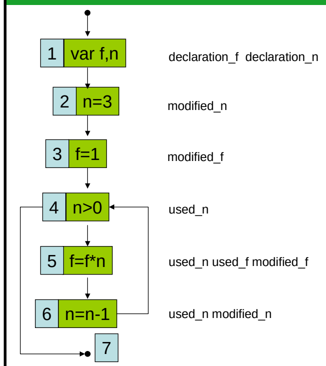


Kripke Structure



So can we model check syntax?

Model Checking Syntax



Yes, e.g.:

AG (delclaration$_f$ $\Rightarrow$ EF used$_f$)

AG (modified$_n$ $\Rightarrow$ EF used$_n$)

also if variables are initialized, certain protocols are respected, locks are released etc.

But: Abstraction is sometimes neither sound nor complete.

The imagination driving Australia's ICT future.
NATIONAL ICT AUSTRALIA

Summary

- model checking syntax is good for finding bugs
- not so good for showing the absence of bugs/verification
- very efficient
- easy to use

The imagination driving Australia's ICT future.
NATIONAL ICT AUSTRALIA

Summary

The imagination driving Australia's ICT future.
NATIONAL ICT AUSTRALIA

Lessons learnt

- “looking” at programs without executing them can reveal a lot of information.
- different static analysis techniques
- application depends on purpose
- combinations with semantic analysis possible (constraint solving, model checking semantics)

NATIONAL ICT AUSTRALIA

Next Week

Model Checking Real-Time Systems