Software and Energy-aware Computing

Fundamentals of static analysis of software

John Gallagher
Roskilde University

ICT-Energy: Energy consumption in future ICT devices

Summer School, Aalborg, Denmark, August 13-16, 2016

Acknowledgements

The partners in the EU ENTRA project (2012-2015)

University of BRISTOL

imdea instituto madrileño de estudios avanzados

XMOS®

UNIVERSITAS ROSKILDENSIS · IN TRANQUILLO MORS · IN FLUCTU VITA ·

Kerstin Eder and team

Pedro López García and team

Henk Muller and team

Roskilde team

http://entraproject.eu

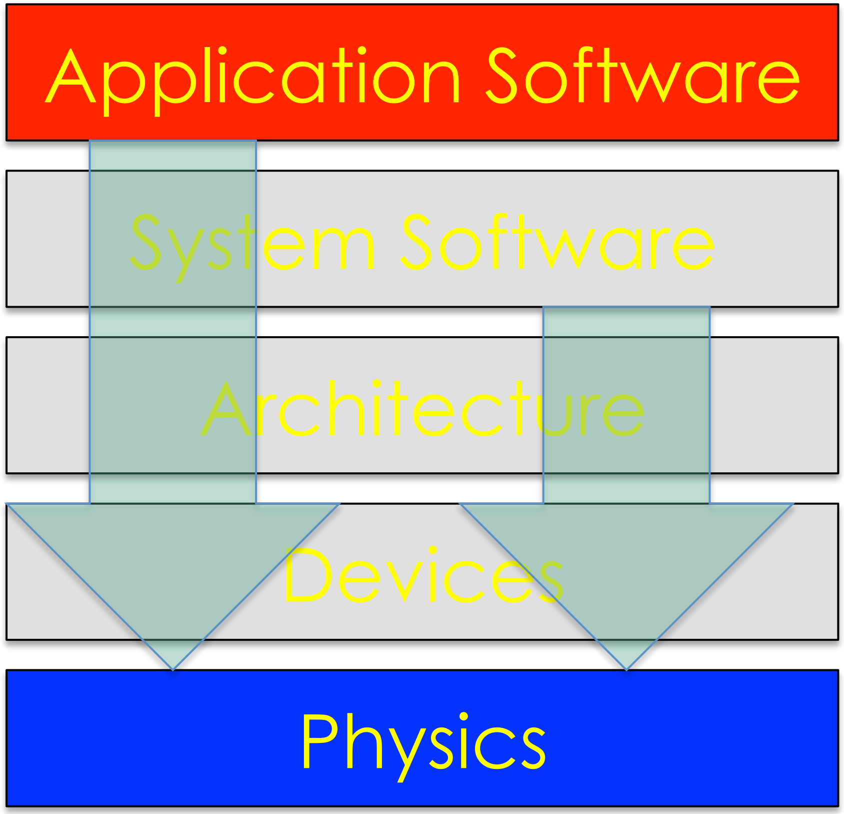
Whole-systems energy transparency

Energy is consumed by **physical processes**.

Yet, application programmers should be able to "see" through the layers and understand energy consumption at the level of code.

The same applies to designers at every level.

How is this possible?



Energy of *software*?

- Energy is consumed by hardware
- But in these lectures we attribute energy cost to software
- Why?
  – (to summarise some of Kerstin's points)

Reason 1

- We take the application programmer's viewpoint
  - programmers don't know much about hardware
  - high-level languages hide the platform from the programmer
    - Which is usually a Good Thing, don't you agree?

Reason 2

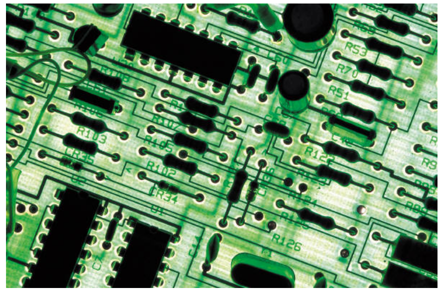
- Energy efficiency as a design goal from the start
- Get an energy profile for a program as early as possible
- Analyse the code to find out how much energy a program will use
- Deliver software with energy guarantees

Reason 2 - continued

- Don't wait to test energy efficiency on hardware, after the software is developed

Development machine

Deployment platform



- It might be too late to fix "energy bugs"

Reason 3

- You can save more energy at the software level than the hardware level

- There are more energy optimisation opportunities higher up the system stack.
- Much energy is wasted by application software

Energy transparency

- Our aim is to let the programmer "see" the energy usage of the code
  - without executing it
  - so that the programmer can "see" where the program wastes energy
  - experiment with different designs

Software factors affecting energy

Important factors are

- Computational efficiency
- Quality of low-level machine code
- Parallelism
- Amount and rate of communication

Computational efficiency

- There is a strong correlation between time and energy consumption (for a single thread)
- Execute as few instructions as possible to achieve the given task, saving energy
- Furthermore, the machine will return more quickly to an idle (low-energy) state

Computational efficiency (2)

- Hence a large part of the energy-aware programmer's job for sequential code is the same as for performance-awareness

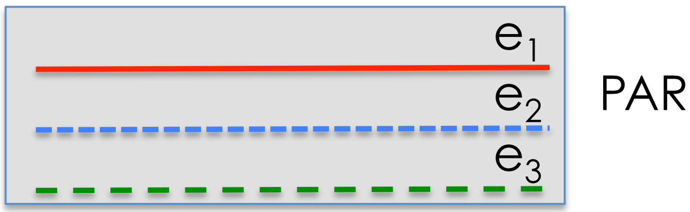
- Get the job done quickly, using efficient algorithms and data structures

Low-level code optimisation

- Given the same high-level code (e.g. C++) there could be many different machine instruction programs.
- Lower energy can be achieved e.g.
  - using VLIW (Very Long Instruction Word) instructions and vectorisation
  - exploitation of low-power processor states using frequency and voltage scaling (DVFS).
- Energy-aware compiler's responsibility

Parallelism

- Is it more energy-efficient to parallelise a task?
- The answer is not straightforward.
- Execution time might be reduced but more energy might be consumed



$e > e_1 + e_2 + e_3$???

If the processors for each process are identical, then the parallel program probably uses more energy.
There is some overhead for managing threads and communication.

Parallelism and clock speed

- Let f = processor clock frequency
- Let P = power
- Let V = voltage
- $\boldsymbol{P = cV^2f}$ (where c is a constant)
- $E = Pt$ (when we run the processor for t time units)
- Hence $e = e_1 + e_2 + \ldots + e_n$ for n processes, if the same total number of instructions is executed, at the same frequency f.

- But if we reduce f, the total energy will reduce because V can also be reduced and P is proportional to V^2!!!

Parallelism (cont'd)

- Hence it is worth parallelising (to save energy) if
  - there is little or no idle time in each processor
    - a waiting processor is wasting energy
  - the clock speed can be reduced in some or all processors, compared to a single process execution

How can static analysis help?

- Automatic complexity analysis
  - understand the best, worst and average cases
  - focus on optimising hot loops
- Timing analysis in multi-threaded code
  - compare parallel algorithm performance, throughput, etc.
  - identify wait times, potential low-power states, etc.

How can static analysis help? (2)

- Analysis of other energy-related resources
  - communication volume and frequency
  - analysis of cache behaviour
  - analysis of memory footprint

SW developer's view

- How do we visualise the results of analysis?
- This is a difficult question in itself.
- Here are some examples and thought experiments

Example

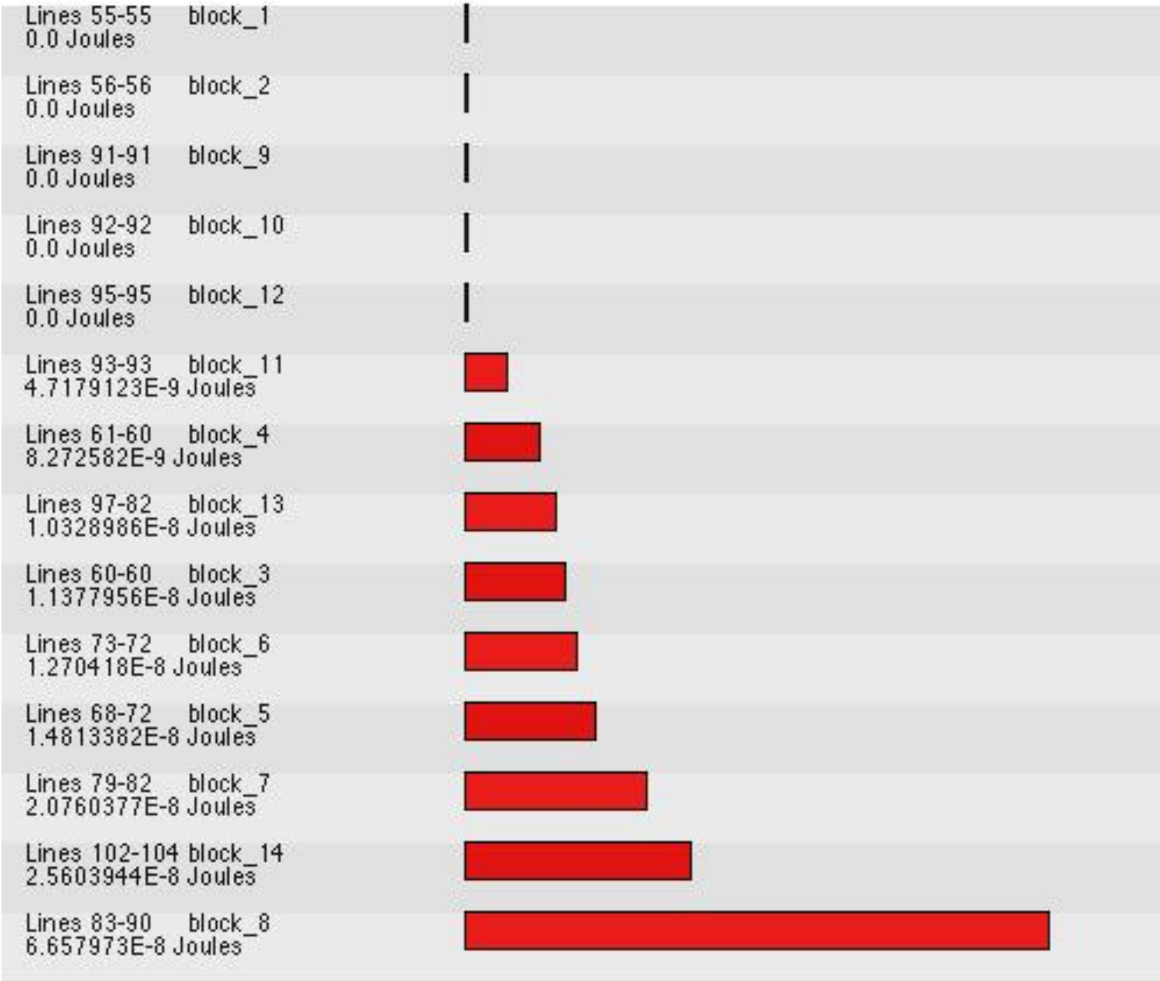
```
#pragma unsafe arrays
int biquadCascade(biquadState &state, int xn) {
    unsigned int ynl;
    int ynh;

    for(int j=0; j<BANKS; j++) {
        ynl = (1<<(FRACTIONALBITS-1));
        ynh = 0;
        {ynh, ynl} = macs( biquads[j].b0, xn, ynh, ynl);
        {ynh, ynl} = macs( biquads[j].b1, state.b[j].xn1, ynh
        {ynh, ynl} = macs( biquads[j].b2, state.b[j].xn2, ynh
        {ynh, ynl} = macs( biquads[j].a1, state.b[j+1].xn1, y
        {ynh, ynl} = macs( biquads[j].a2, state.b[j+1].xn2, y
        if (sext(ynh,FRACTIONALBITS) == ynh) {
            ynh = (ynh << (32-FRACTIONALBITS)) | (ynl >> FRAC
        } else if (ynh < 0) {
            ynh = 0x80000000;
        } else {
            ynh = 0x7fffffff;
        }
        state.b[j].xn2 = state.b[j].xn1;
        state.b[j].xn1 = xn;

        xn = ynh;
    }
    state.b[BANKS].xn2 = state.b[BANKS].xn1;
    state.b[BANKS].xn1 = ynh;
    return xn;
}
```

biquadCascade(BANKS) = 157 * BANKS + 51.7 nJoules

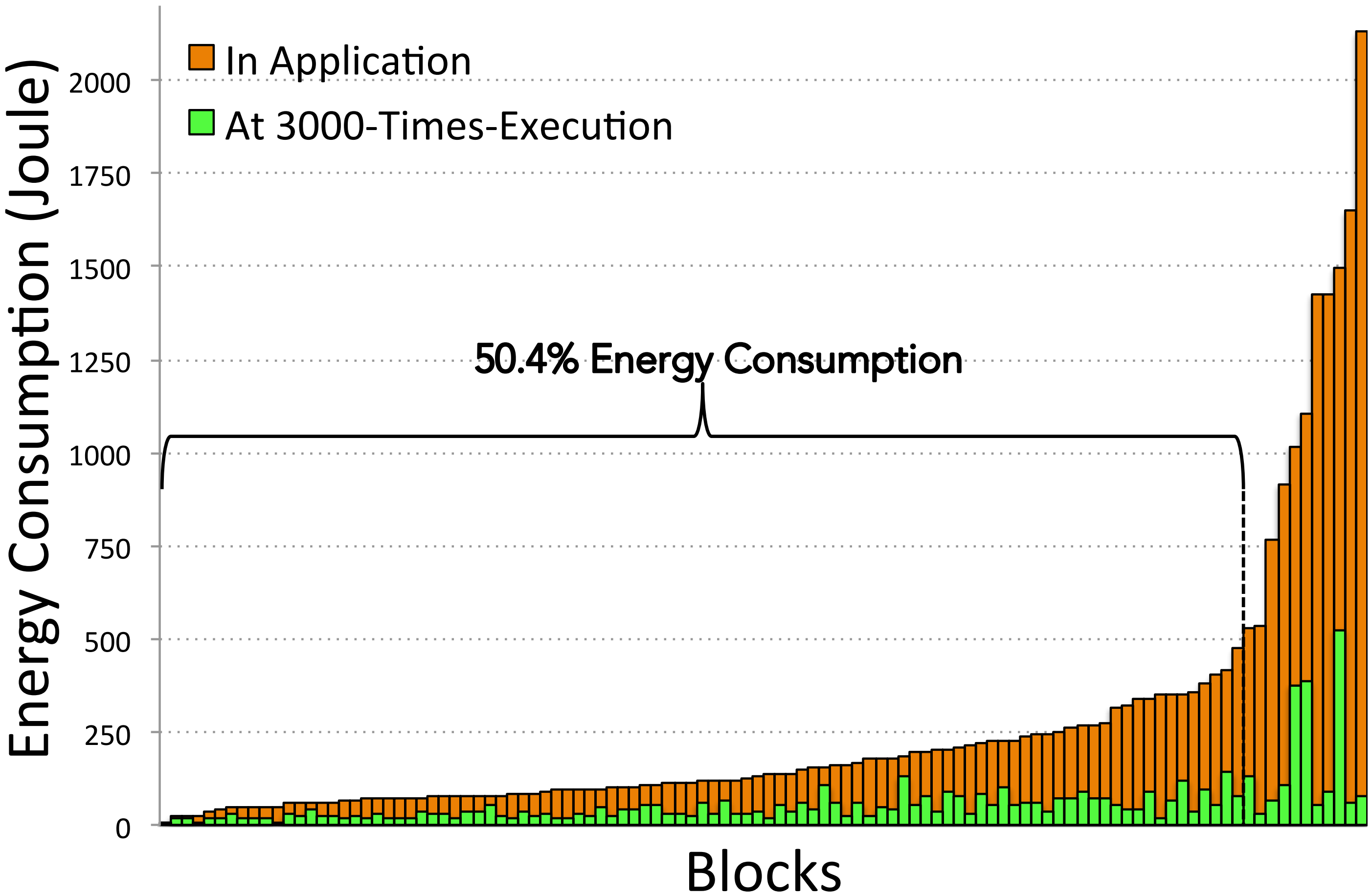
This is an estimate of the energy used by the function.

It is a linear function of the value of BANKS

Visualise energy of program blocks



Which code blocks are hot?



Example

```
in port inP = XS1_PORT_4A;
out port led_port = XS1_PORT_1E;

void consumer(chanend couts) {
      int j;
      while (1) {
            couts :> j;
            for (int i=0;i<j;i++)
                  led_port <: (i & 1);
      }
}
```

12.3%

72.4%

Simulation with random 0..15 values on input port.

```
void producer(int n, chanend couts) {
      for (int i=0;i<n;i++) {
            printf("i=%d\n",i);
            couts <: i;
      }
}

int main () {
      chan a;  int x;
      par {
            while (1){
                  inP :> x;
                  producer(x,a);
            }
            consumer(a);
      }
}
```

13.8%

1.5%

Energy a design goal for programmers

```
#pragma check energy(proc(x))<5pJ
int proc(int x) {
...
```

```
Output:
Checked 0≤x≤5⇒energy(proc(x))<5pJ
```

Summary of goals

- Tools for the programmer
  - that give information about the energy usage of programs without running them (energy transparency)
  - that allow energy assertions to be checked (energy design goals)

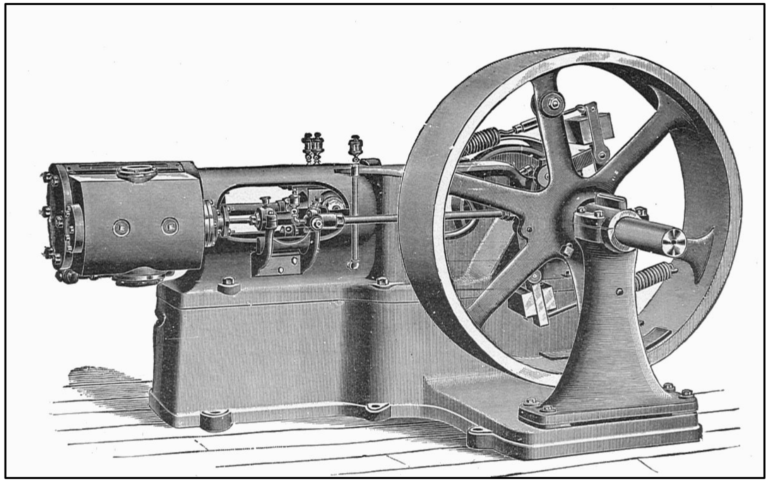
Semantics and program analysis

- To achieve the goals we need tools for program analysis

- Program analysis is based on formal program semantics
  - the mathematical study of program meanings

Programs are machines (that consume energy)

```
n = 4;
z = 1;
while (n > 0) {
    z = z*n;
    n = n-1;
}
print(z);
```



Semantics gives the "machine" defined by a program.

Analysis of programs

- A program is a physical object. e.g.
  - some symbols on paper
  - a pattern of bits in memory
- But what is the meaning of a program?
- This is program semantics.

Tiwari's Energy Equation (from Kerstin's slides)

$$E_P = \sum_i (B_i \times N_i) + \sum_{i,j} (O_{i,j} \times N_{i,j}) \; .$$

- N_i is the number of times instruction *i* is executed.
- $N_{i,j}$ is the number of times instruction *i* is followed by instruction *j* in the program execution.
- The aim of static analysis is to determine N_i and $N_{i,j}$ <u>for all possible program executions</u>

Program semantics

```
n = 4;
z = 1;
while (n > 0) {
    z = z*n;
    n = n-1;
}
print(z);
```

To execute or analyse
this program,
we need to understand
the meaning of teh symbols
such as “while”, “>”, “*”,
“;”, ”{”, “}”, etc.

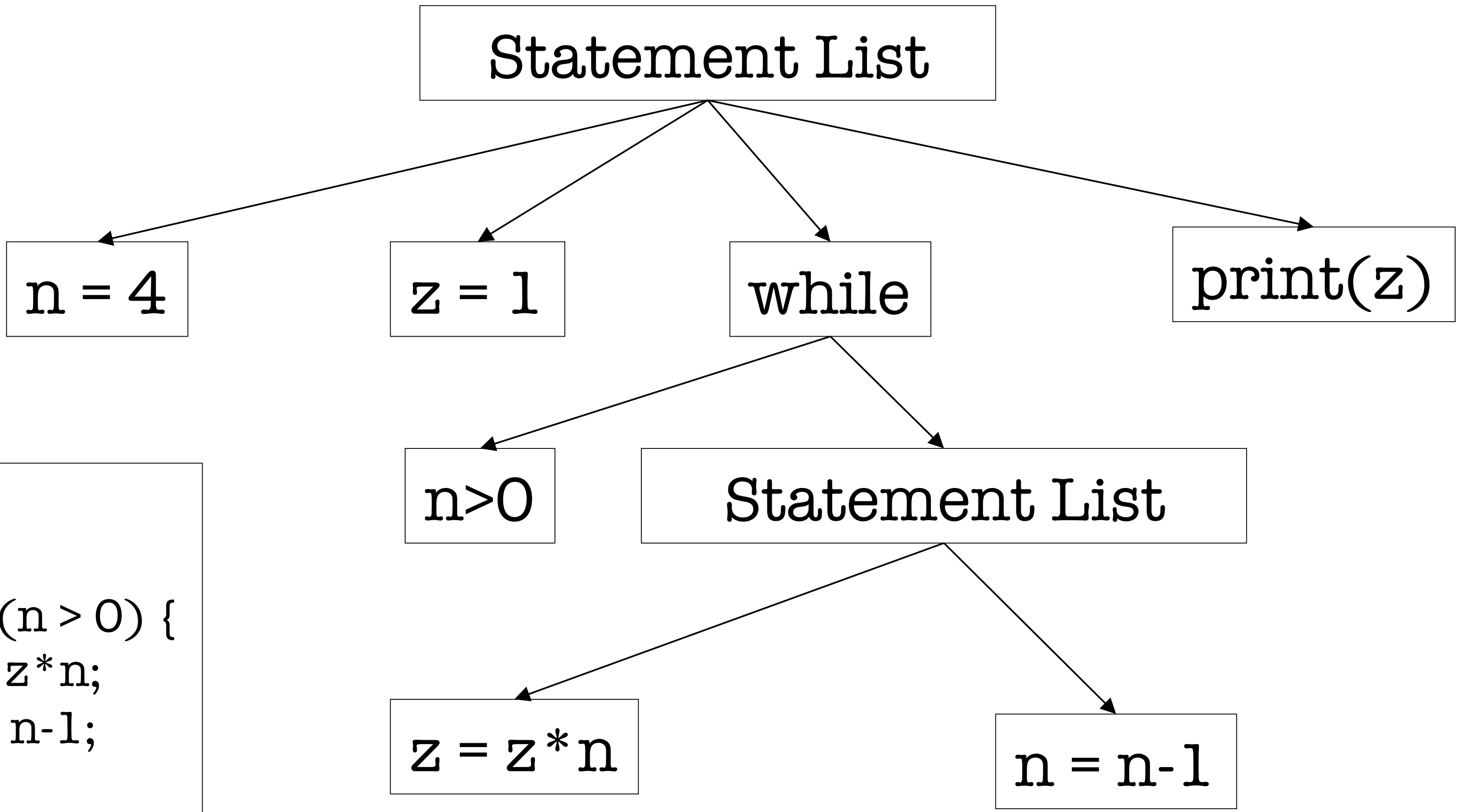
Different styles of program semantics

- Operational semantics
  - **small steps** (from one state to the next)
  - big steps (from the start to the end state)
  - Hoare-Floyd conditions
- Denotational semantics
  - the mathematical function represented by a program
  - obtained by composing the functions representing its parts

Phases of semantic analysis

1. Syntax analysis (parsing)
   - breaking the program into is basic parts and determining its structure
2. Semantic translation
   - representation of the program in some suitable mathematical or logical form
3. Semantic interpretation
   - using the semantic representation to analyse the program execution

Program syntax tree (parsing)

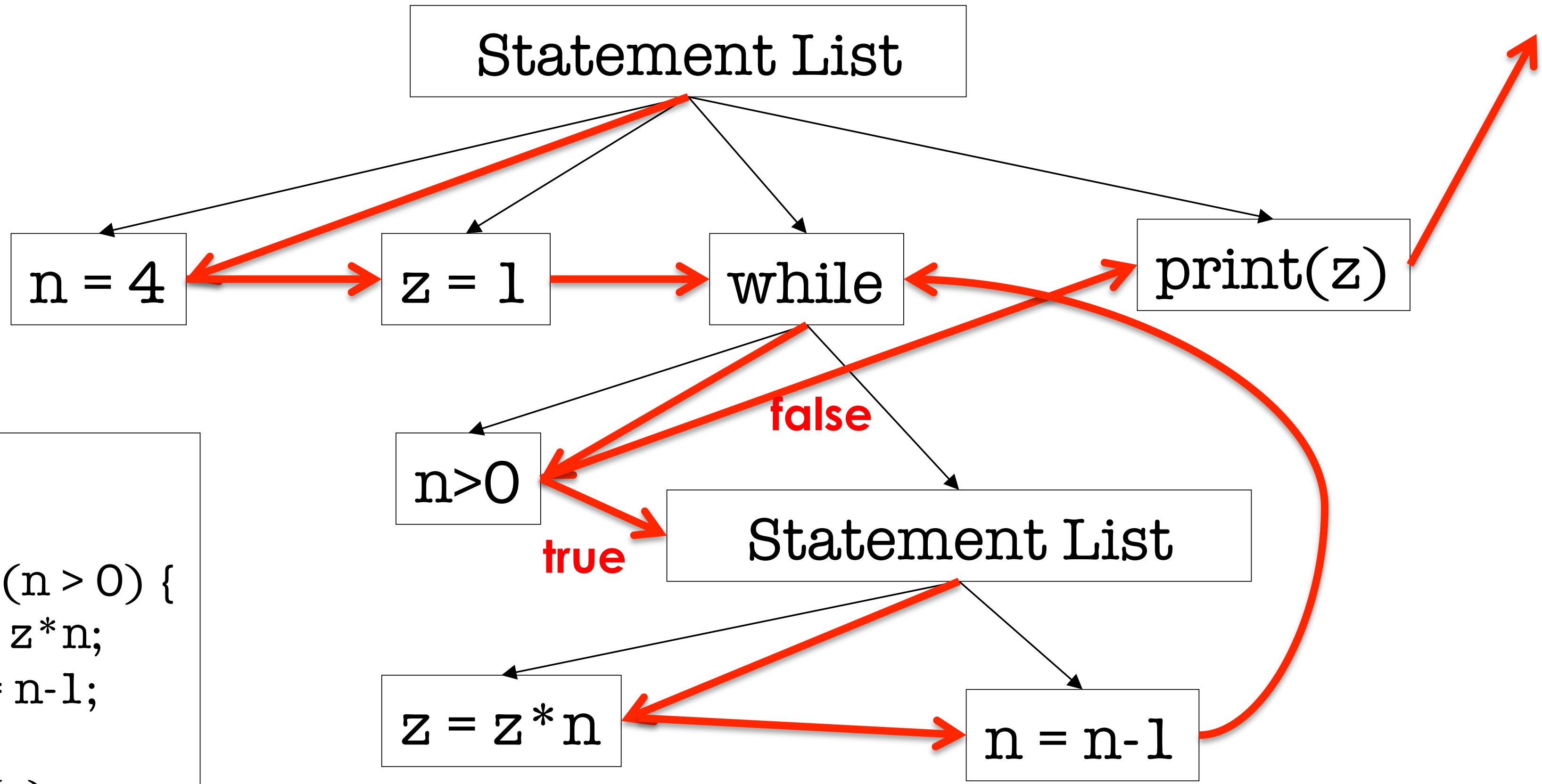



```
n = 4;
z = 1;
while (n > 0) {
    z = z*n;
    n = n-1;
}
print(z);
```

From syntax tree to flow graph

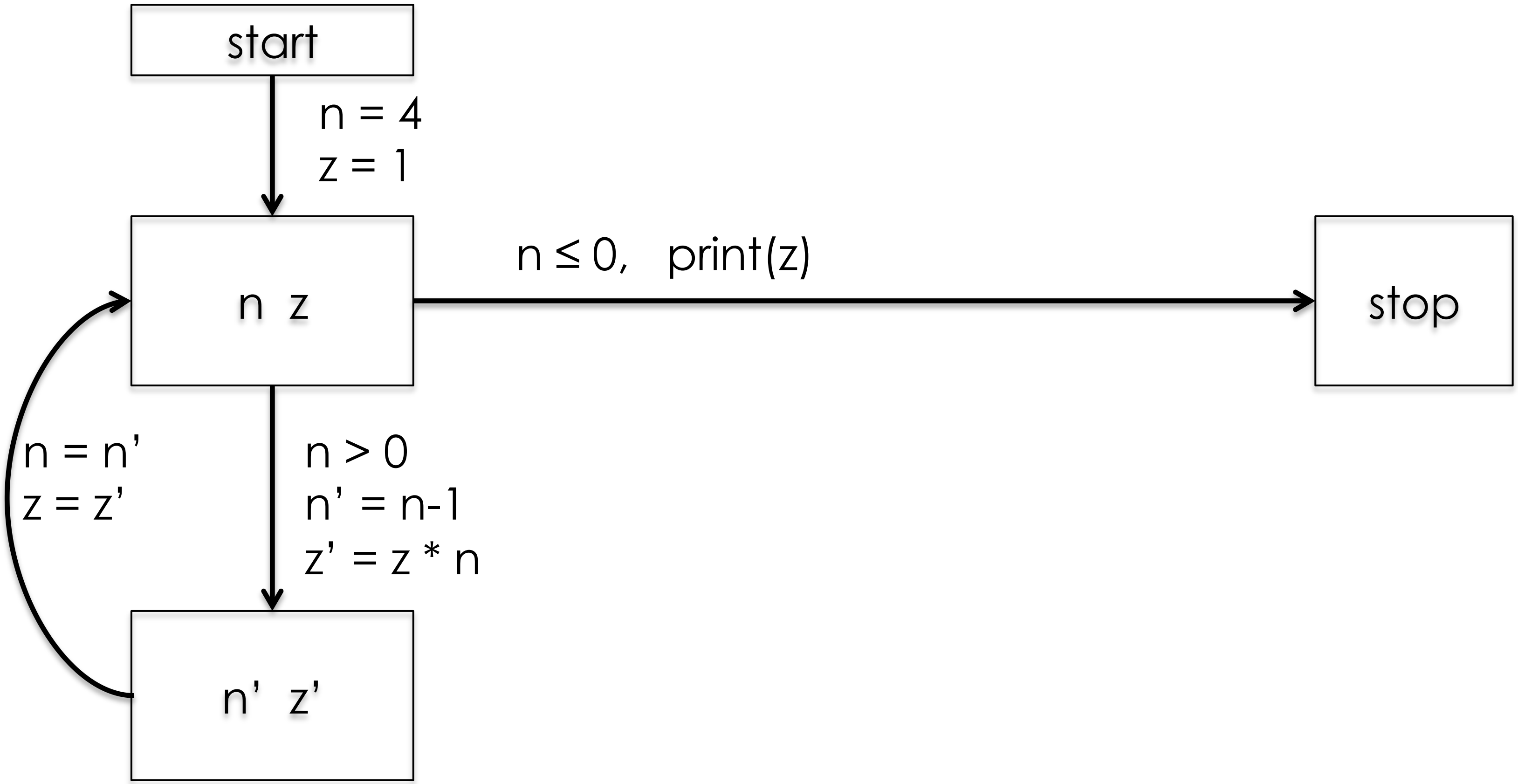
Grammar Rules	Semantic Rules for flow of control
If → if E then S_1 else S_2	E.true := S_1 E.false := S_2 S_1.next := If.next S_2.next := If.next
While → while E S_1	E.true := S_1 E.false := While.next S_1.next := While
StatementList → $S_1 S_2$ ..... S_n	S_j.next = S_{j+1} (j = 1 to n-1) S_n.next := StatementList.next
S → StatementList | If | While | Print | Assign	StatementList.next := S.next If.next := S.next While.next := S.next Print.next := S.next Assign.next := S.next

From syntax tree to flow graph




```
n = 4;
z = 1;
while (n > 0) {
    z = z*n;
    n = n-1;
}
print(z);
```

From flow graph to state automata



Exercise

1. Draw the syntax tree
2. Draw the control flow graph
3. Draw the state automaton

```
while (m != n) {
    if (m > n) {
        m = m-n;
    }
    else {
        n = n-m;
    }
}
```

Phases of semantic analysis

1. Syntax analysis (parsing)
   – breaking the program into is basic parts and determining its structure
2. Semantic translation
   – representation of the program in some suitable mathematical or logical form
3. Semantic interpretation
   – using the semantic representation to analyse the program execution

From automaton to predicate logic

Horn clauses

$\text{true} \rightarrow \text{reachable}_1$

$(\text{reachable}_1 \wedge n=4 \wedge z=1)$

$\rightarrow \text{reachable}_2(n,z)$

$(\text{reachable}_2(n,z) \wedge n<0 \wedge z'=z*n \wedge n'=n-1)$

$\rightarrow \text{reachable}_3(n',z')$

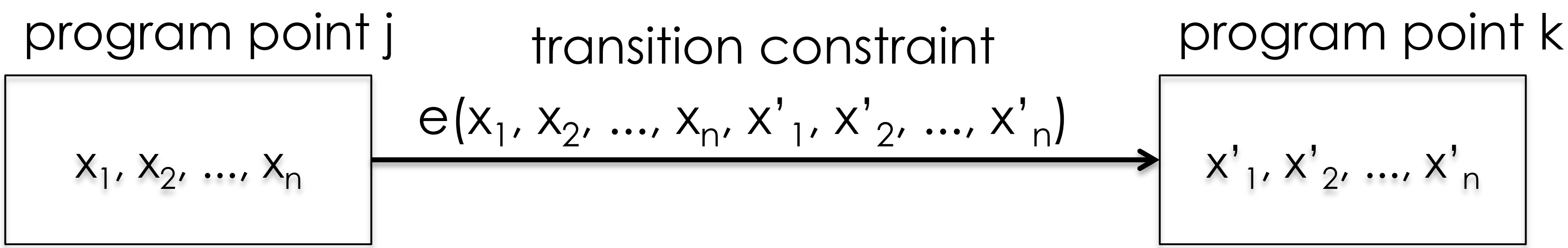
$(\text{reachable}_3(n',z') \wedge n=n' \wedge z=z')$

$\rightarrow \text{reachable}_2(n,z)$

$\text{reachable}_2(n,z) \wedge n \geq 0 \wedge \text{print}(z))$

$\rightarrow \text{stop}$

Logical representation



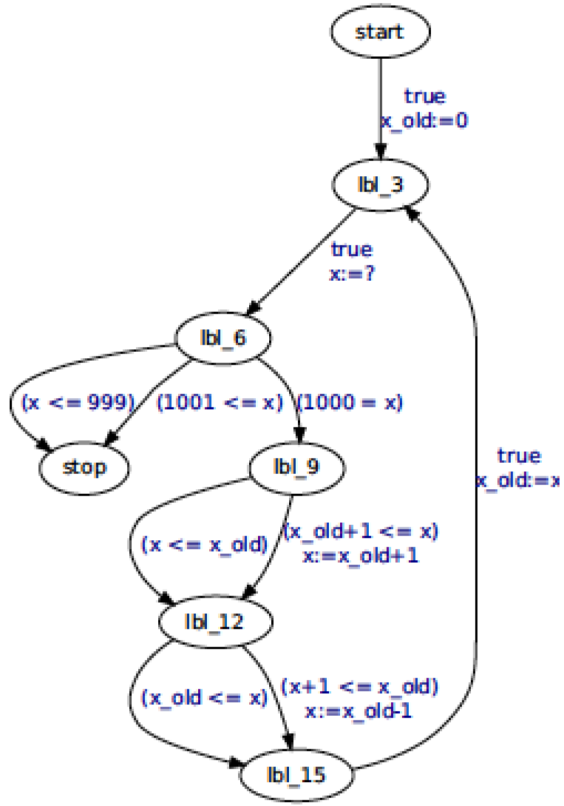
$$(\mathrm{reachable}_j(x_1, x_2, \ldots, x_n) \wedge e(x_1, x_2, \ldots, x_n, x'_1, x'_2, \ldots, x'_n)) \rightarrow \mathrm{reachable}_k(x'_1, x'_2, \ldots, x'_n)$$

Example: A rate limiter*

*Example by Monniaux

Listing 5. Rate limiter

```
void main() {
  float x_old, x;
  x_old = 0;
  while (1) {
    x = input(-1000,1000);
    if (x >= x_old+1)
        x = x_old+1;
    if (x <= x_old-1)
        x = x_old-1;
    x_old = x;
  }
}
```

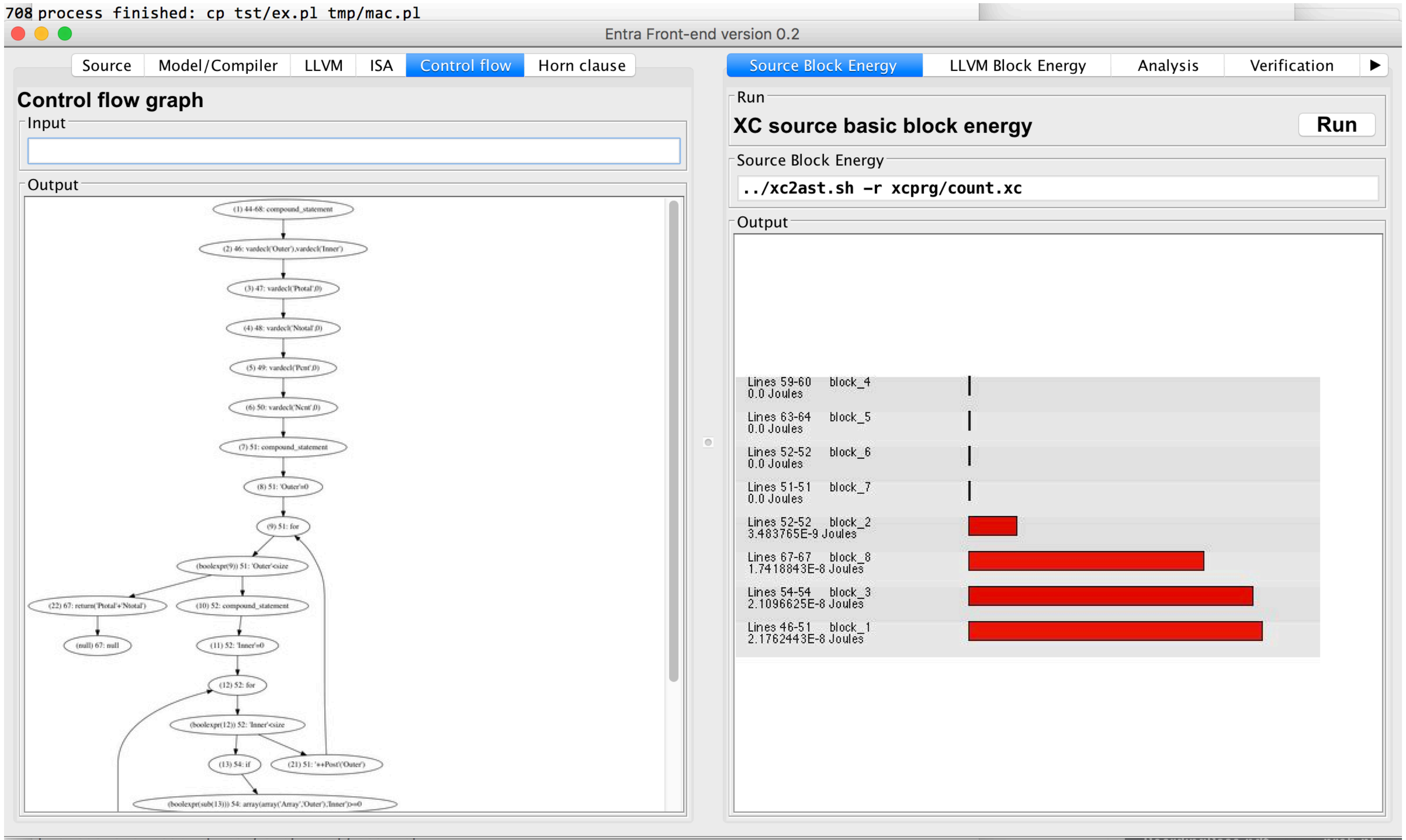


Rate limiter – logic representation

```
r1(X,X_old) :-
        X_old=0,
        r0(_,_).
r1(X,X_old) :-
        r5(X,X_old).

r2(X,X_old) :-
        X >= -1000,
        X =< 1000,
        r1(_,X_old).

r3(X,X_old) :-
        X1 >= X_old+1,
        X = X_old+1,
        r2(X1,X_old).
```

```
r3(X,X_old) :-
        X < X_old+1,
        r2(X,X_old).

r4(X,X_old) :-
        X1 =< X_old-1,
        X = X_old-1,
        r3(X1,X_old).
r4(X,X_old) :-
        X > X_old-1,
        r3(X,X_old).

r5(X,X_old) :-
        X_old=X,
        r4(X,_).
```

More examples from ENTRA tool



Identification of basic blocks

- A basic block is a section of “straight-line” code.
  - The start of a block is a branch or merge point
  - The end of a block is a branch or jump
- Basic blocks can be extracted from the control flow graph
- Every statement in a basic block is executed the same number of times

Phases of semantic analysis

1. Syntax analysis (parsing)
   - breaking the program into is basic parts and determining its structure
2. Semantic translation
   - representation of the program in some suitable mathematical or logical form
3. Semantic interpretation
   - using the semantic representation to analyse the program execution

Program analysis

- Program properties
- Program invariants
- Global properties that depend on summary of an infinite number of behaviours

- Prove absence of bugs (verification) rather than presence (testing/ simulation)

Invariants

- Many program analysis and verification tasks involve proving invariants
- An invariant is an assertion that is true at a given program point.
- We consider invariants on energy usage.

Example invariant

```
void main() {
  float x_old, x;
  x_old = 0;
  while (1) {
    x = input(-1000,1000);
    if (x >= x_old+1)
        x = x_old+1;
    if (x <= x_old-1)
        x = x_old-1;
    x_old = x;
  }
}
```

Check assertion

$-1000 \leq x\_old \leq 1000$

Proving invariants

- To prove that invariant P holds at program point j, prove the following implication

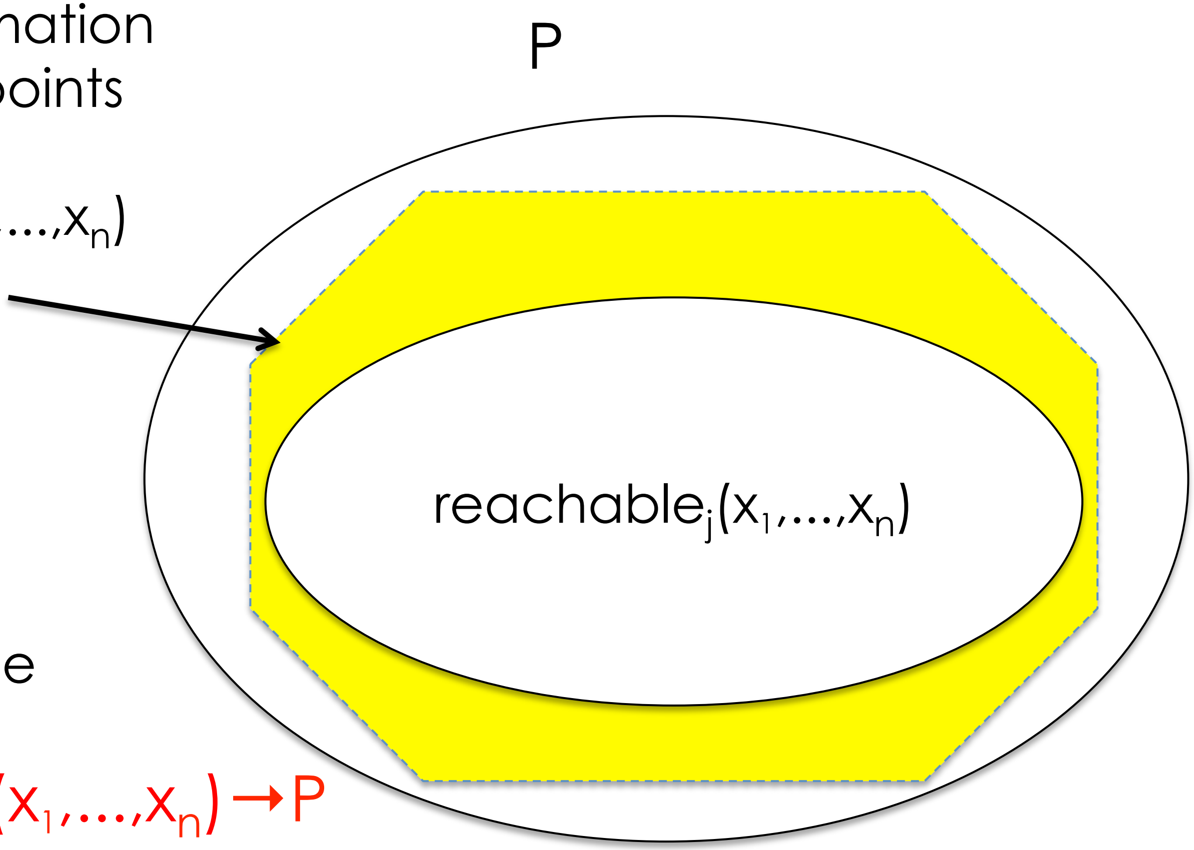
$\text{reachable}_j(x_1,\ldots,x_n) \rightarrow P$

which is equivalent to

$\neg(\text{reachable}_j(x_1,\ldots,x_n) \wedge \neg P)$

Proof by approximation

Overapproximation of the set of points where $reachable_j(x_1,\ldots,x_n)$ is true.



Contained within P, hence

$reachable_j(x_1,\ldots,x_n) \rightarrow P$

Energy invariants

- The program state can contain resource counters.
- $reachable_k(x_1,\ldots,x_n,e)$ means that the total energy consumed is e, when the program reaches point k

- So we can express and prove assertions about energy (or other resources)
- More on this later...

Two basic techniques

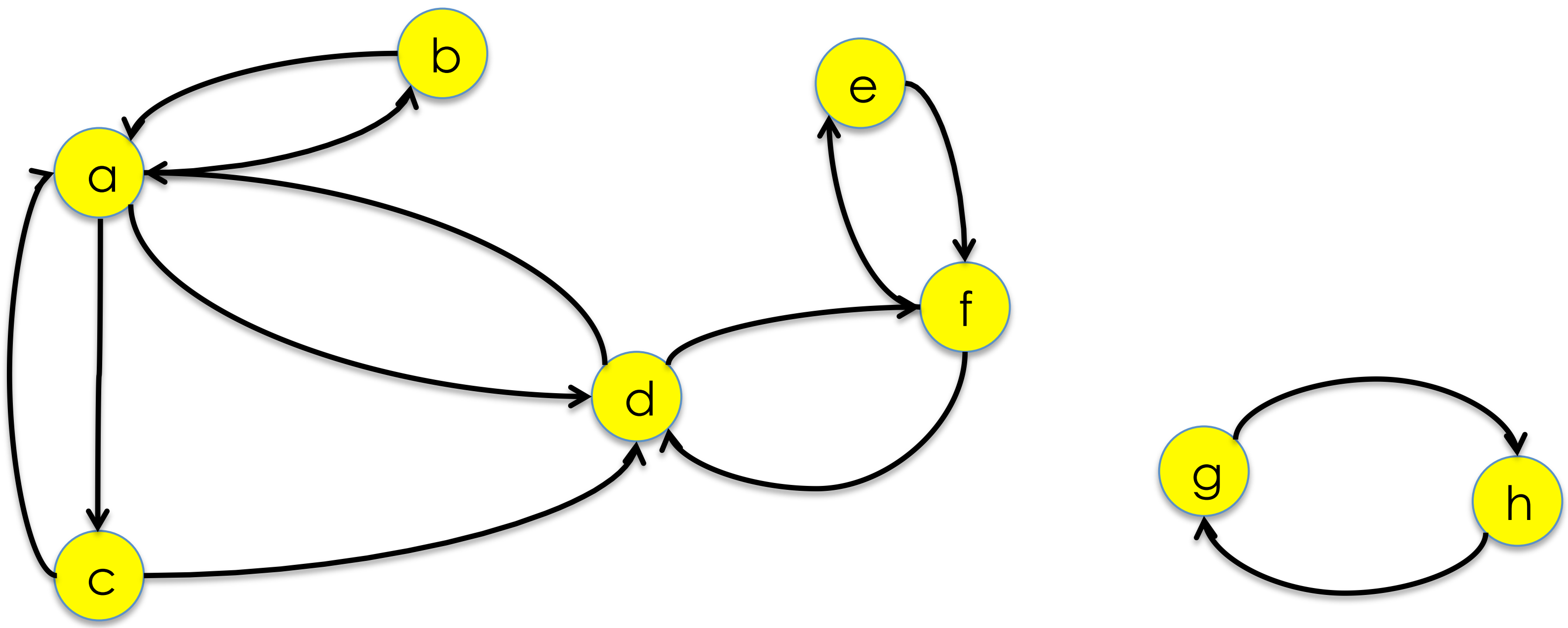
- How to capture all reachable states?
  - answer, fixpoint techniques
- How to capture an infinite set of states?
  - answer, abstract interpretation
- These two methods underlie much program analysis

Fixpoint computation

- Sounds complicated, but it is a very simple procedure
- It is a closure or saturation procedure

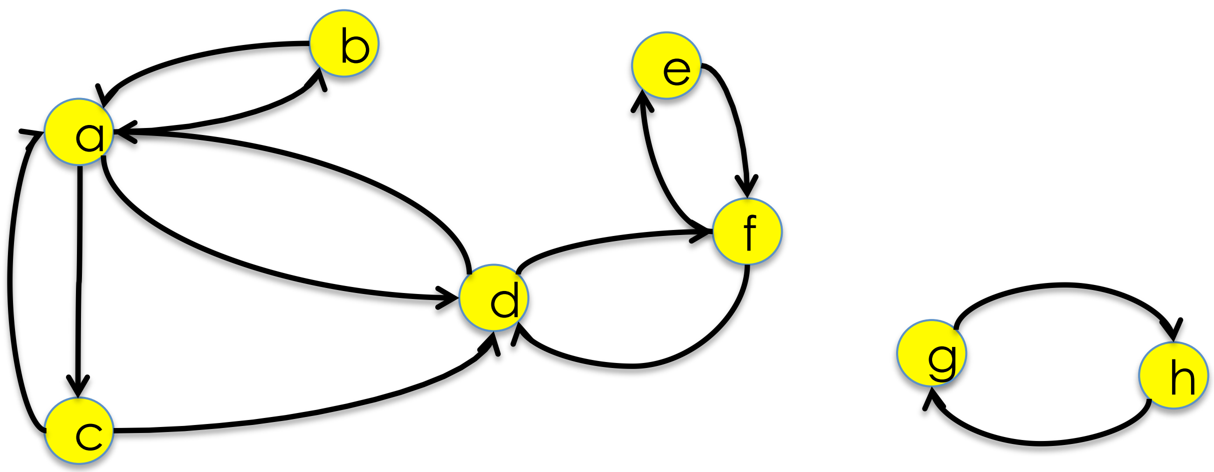
Fixpoint example

- Consider a route network, with stations a,b,...,h



post(S) function

- Let S be a set of stations.
- post(S) is the set of stations reachable in one step from S. E.g. post({a,h}) = {b,c,d,g}



Reachability as a fixpoint

- The set of stations reachable from an initial set S, called Reach(S) is defined as the smallest set Z such that **$Z = F(Z)$**

  where $F(Z) = S \cup post(Z)$

- This can be computed as the limit of a sequence $\emptyset, F(\emptyset), F(F(\emptyset)), \ldots$

Example

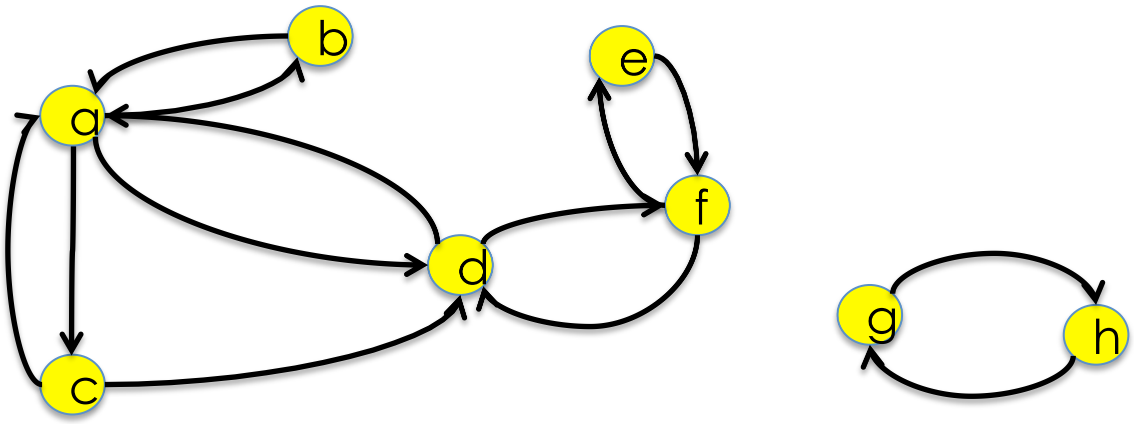
- Find the stations reachable from a.

F(Z) = {a} ∪ post(Z)

∅

F(∅) = {a}
F({a}) = {a,b,c,d}
F({a,b,c,d}) = {a,b,c,d,f}
F({a,b,c,d,f}) = {a,b,c,d,e,f}
F({a,b,c,d,e,f}) = {a,b,c,d,e,f}

fixpoint found {a,b,c,d,e,f}

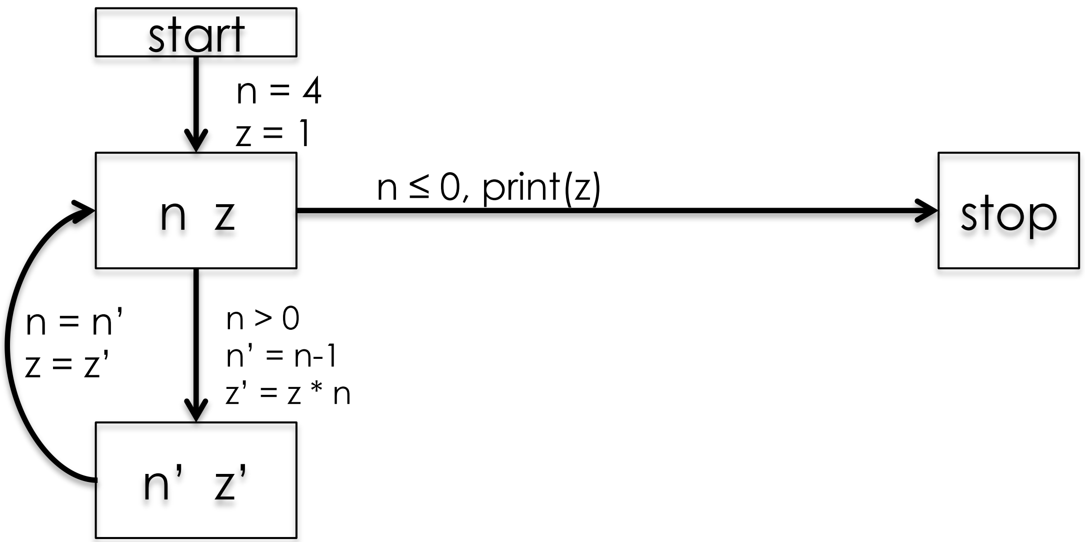


Exercise

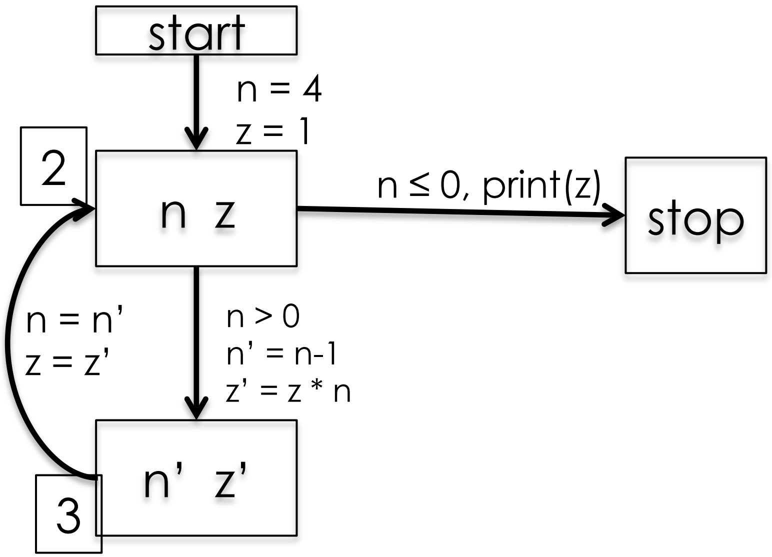
- Using the same graph, compute the set of states reachable from e, using a fixpoint computation.

The reachable states of a program

- We apply the same idea to find the reachable states of a program, starting with the initial state.



The reachable states of a program



2	3
{}	{}
{(4,1)}	{}
{(4,1)}	{(3,4)}
{(4,1),(3,4)}	{(3,4)}
{(4,1),(3,4)}	{(3,4),(2,12)}
….	….
{(4,1),(3,4), (2,12),(1,24), (0,24) }	{(3,4),(2,12),(1,24)}

(n,z) represents the values of n and z at a given point

Infinite fixpoints

- However, usually the set of reachable states of a program is infinite, and the sequence could keep on growing
- We might never reach the fixpoint

- In this case we use abstraction

Abstract interpretation

Example

- 476305 × -576 = 274351680

- Is the above equation correct?

Rule of signs

- The rule of signs is an abstraction of the multiplication relation

$$+ \times + = +$$
$$+ \times - = -$$
$$- \times + = -$$
$$- \times - = +$$

We can check incorrectness, but not correctness with the rule of signs.

The interval abstraction

- The value of a variable is abstracted by an interval
  - The variable has any value within the interval
- We can perform operations on intervals, as we did for signs

- E.g. [3,10] + [-2,6] = [3+(-2), 10+6] = [1,16]

- Exercise. What is [3,10] – [-2,6]?

Example: interval abstraction

- The set of pairs of values {(4,1),(3,4), (2,12),(1,24),(0,24) } can be abstracted by the pair of intervals ([0,4], [1,24])
- So n is between 0 and 4, z is between 1 and 24.
- But information has been lost
  - the pair (3,19) is also consistent with the intervals.
  - the intervals give an over-approximation of the reachable states.

Convex polyhedra

- A more precise abstraction than intervals is given by convex polyhedra
- Convex polyhedra are linear inequalities among the state variables

Example convex polyhedron abstraction

```
var i,j:int;
begin
  i=0; j=10;
  while i<=j do
    i = i+2;
    j = j-1;
  done;
end
```

```
r1(I,J) :-
     I=0,J=10.
r2(I,J) :-
     r1(I,J).
r2(I,J) :-
     I1 =< J1,
     I = I1+2,
     J = J1-1,
     r2(I1,J1).
r3(I,J) :-
     I >= J+1,
     r2(I,J).
```

Approximate reachable states

```
r1(I,J) = [I=0,J=10].
r2(I,J) = [-I >= -16,I >= 0,I+2*J=20].
r3(I,J) = [-3*I >= -26,3*I >= 22,I+2*J=20].
```

This result is computed fast, using the Parma Polyhedra Library to perform the operations on convex polyhedra.

Summary so far....

- We can translate a program to a state automaton
- We can compute over-approximation of the reachable states of the program
  - using fixpoint computation and abstraction
- We can use the approximation to check assertions about the program.