Syntax Directed Translation to Low-Level VMcode

Armando Solar-Lezama and Mike Carbin

Status so far

Semantics as recursive inference rules

"Easy" to map directly to a recursive interpreter

- Unstructured control flow can be a little messy, but implementation directly matches semantics

Very inefficient!

- Interpreter is continuously traversing a large datastructure
- Deep recursive calls have high overheads
- A simple statement like x = 2+2 may involve thousands of instructions

Option 1: Compile to machine code

Plus:

- Entirely avoid overhead of interpretation

Minus:

- Generating machine code can be slow
- Machine code needs to be tailored to specific OS/Architectures

Low-Level VM

Provide a layer of abstraction between the code and hardware

Faster to interpret than the original AST

Easier to compile from relative to AST

When we are ready to generate machine instructions, it will be easier

Portable!

General Characteristics

Simple instructions perform 1 operation at a time

Explicit control flow through jumps

More explicit addressing of storage

Translation to LL instructions

The translation to low-level instructions must do the following

- Make control flow explicit through jumps
- Break complex expressions into sequences of simple instructions
- Associate variable names with explicit storage locations

Two step process:

- AST -> CFG
- CFG -> Instructions

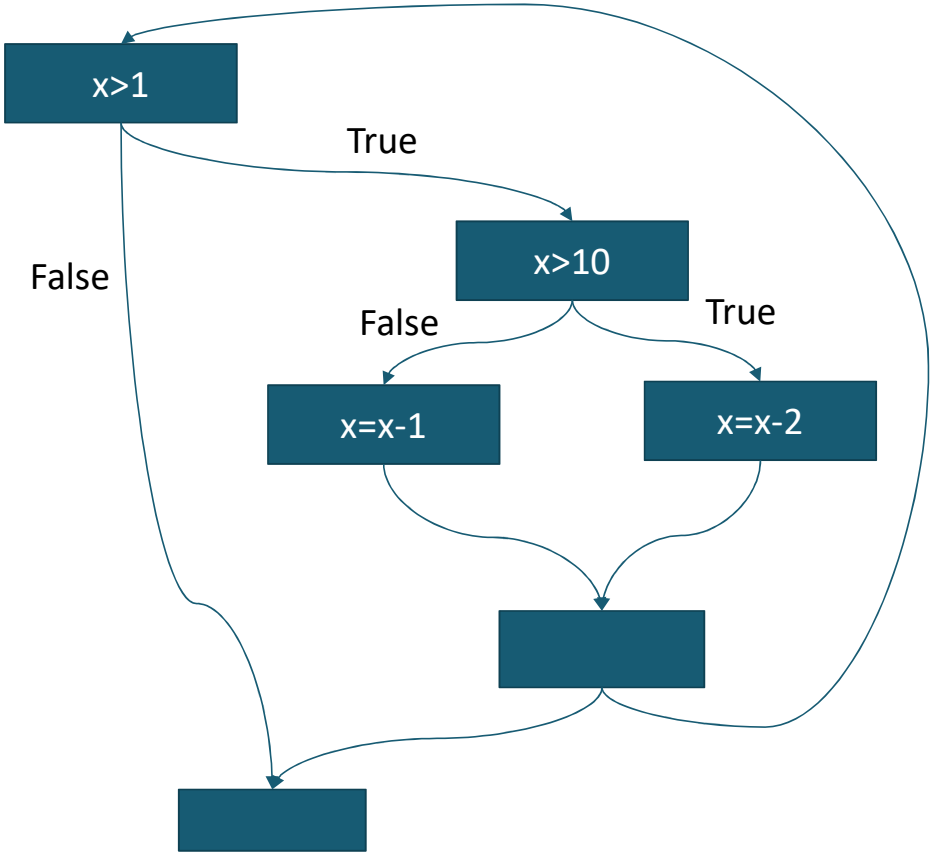
Deduction rules for translation

$$\frac{Premises}{Statement \rightarrow Representation}$$

Representing Control Flow

Control flow graph

```
while(x>1){
    if( x > 10){
        x = x – 2;
    }else{
        x = x – 1;
    }
}
```



CFG Formally

(BB, I, F, E)

- BB is a set of basic blocks
  - A basic block is a sequence of instructions that must always execute together
  - Execution always starts with the first instruction and if the first instruction executes, all instructions in the basic-block execute.
  - The last instruction may be a predicate, which determines which outgoing edge to follow
- $I \in BB$ is the entry point of the CFG
- $F \in BB$ is the exit point of the CFG
- $E \subset BB \times L \times BB$ is a set of edges potentially with a label
  - By construction, we ensure that every basic block with outgoing edges has either one unlabeled outgoing edge, or two outgoing edges with labels true and false respectively.

CFG Construction Try 1

First we will ignore return statements

$$\frac{Premises}{Statement \rightarrow (BB, I, F, E)}$$

Simple language of statements

S := if(e) S1 else S2

while(e) S1

S*

x = e

Syntax directed translation rules

$$\frac{I = F = [x = e] \quad BB = \{I\} \quad E = \{\}}{x = e \ \rightarrow (BB, I, F, E)}$$

$$\frac{\begin{array}{c} S1 \rightarrow (BB_1,\ I_1, F_1, E_1) \quad S2 \rightarrow (BB_2, I_2, F_2, E_2) \\ I = [e] \qquad F = [noop] \\ BB = BB_1 \cup BB_2 \cup \{I, F\} \\ E = E_1 \cup E_2 \cup \{(I, True, I_1), (I, False, I_2), (F_1, \perp, F), (F_2, \perp, F)\} \end{array}}{if\,(e) S1\ else\ S2 \rightarrow (BB, I, F, E)}$$

$$\frac{\begin{array}{c} S1 \rightarrow (BB_1,\ I_1, F_1, E_1)\ \ I = [e] \qquad F = [noop] \qquad BB = BB_1 \cup BB_2 \cup \{I, F\} \\ E = E_1 \cup \{(I, True, E_1), (I, False, F), (F_1, \perp, I)\} \end{array}}{while(e)\ S1\ \rightarrow (BB, I, F, E)}$$

Syntax directed translation rules

$$S \rightarrow (BB_1,\ I_1, F_1, E_1)$$
$$rest \rightarrow (BB_2, I_2, F_2, E_2)$$
$$BB = BB_1 \cup BB_2$$
$$I = I_1 \qquad F = F_2$$
$$\frac{E = E_1 \cup E_2 \cup \{(F_2, \perp, I_1)\}}{S{:}\,rest\ \rightarrow (BB, I, F, E)}$$

Merging basic blocks

The earlier rules are simple, but they lead to too many basic blocks

If you have two basic blocks B1 and B2 where

- B1 has only one unlabeled outgoing edge to B2
- B2 has only one unlabeled incoming edge from B1

You can merge B1 and B2 into a single block

This can be done easily as the graph is being constructed

- The rule for S* would be responsible for this

What about return?

Unstructured control flow

- any construct that leads a construct to have more than one exit point or entry point
- MITScript only has return
- other common constructs include
  - continue
  - break
  - throw

We can deal with this by adding some context to our rules

$$\frac{Premises}{(Context, Statement) \rightarrow (Context, Representation)}$$