Utrecht University

Concepts of Programming Language Design

Abstract Machines

Gabriele Keller
Tom Smeding

Abstract Machines

- What is an abstract machine?
  - a set of legal states
    - final and initial states as subset
  - A set of instructions altering the state of the machine
    - it should be possible to implement the operations on a real machine in a finite (preferably constant) number of steps
- Why use abstract machines at all?
  - specifies the semantics of a programming languages
  - can specify architecture independent implementation
- We have seen this before
  - similar to SOS, but with abstract machines, we care about performance

Utrecht University

Control Flow

- Base line: the M-machine
  - small step semantics for MinHs
  - transition system embodies essentially a very high-level (= concise, but inefficient) abstract machine
  - we'll call it the M-machine from now on, to distinguish it from other abstract machines for MinHs
- Characteristics of the M-machine
  - substitution as "machine operation"
    - why is that not inefficient?
    - can be avoided by using an environment
  - control-flow is not explicit
    - the search rules determine next subexpression to be evaluated
    - why is that inefficient?

Control Flow

- Example:

$$\frac{}{\texttt{(Plus (Num } n\texttt{) (Num } m\texttt{))} \mapsto \texttt{(Num (}n\texttt{ + }m\texttt{))}}$$

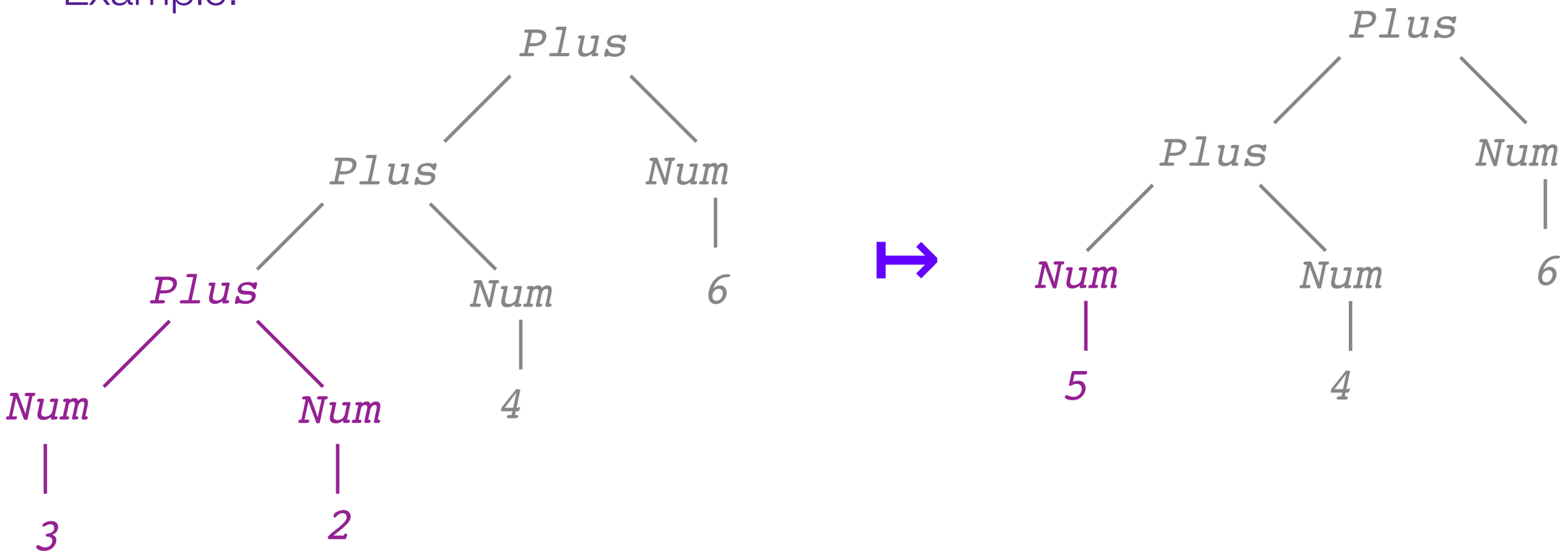
$$\frac{e_2 \mapsto e_2'}{\texttt{(Plus (Num } n\texttt{) } e_2\texttt{)} \mapsto \texttt{(Plus (Num n) } e_2'\texttt{)}}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{(Plus } e_1\ e_2\texttt{)} \mapsto \texttt{(Plus } e_1\ e_2'\texttt{)}}$$

Utrecht University

Control Flow

- Example:



$$\frac{\texttt{Plus(Num 3)(Num 2)} \mapsto \texttt{Num 5}}{\texttt{Plus(Plus(Num 3)(Num 2))(Num 4)} \mapsto \texttt{Plus(Num 5)(Num 4)}}$$

$$\texttt{Plus(Plus(Plus(Num 3)(Num 2))(Num 4))(Num 6)} \mapsto \texttt{Plus(Plus(Num 5)(Num 4))(Num 6)}$$

depending on the size & nesting depth of the expression, searching for the next reducible subexpression can be very expensive!!!

Control Flow

- Single-step evaluation in Haskell:

```
data Expr
  = Num   Int
  | Plus  Expr Expr
  | Times Expr Expr
```

```
bigStep :: Expr -> Int
bigStep (Num n)
  = n
bigStep (Plus e1 e2)
  = (bigStep e1) + (bigStep e2)
```

```
single (Plus (Num n1) (Num n2)) =
  Num (n1 + n2)
single (Plus (Num n1) e) =
  Plus (Num n1) (single e)
single (Plus e1 e2) =
  Plus (single e1) e2
```

```
n
 (single e)
```

- for each step, the expression has to be traversed to find the next subexpression that has to be evaluated
- makes heavy use of Haskell's runtime stack

The C-machine

- Explicit control flow: C-machine
  - explicit stack
  - explicit handling of control flow
  - variable binding still handled by substitution
  - we call this machine the C-machine
- Machine state
  - the current expression (as before)
  - a control stack of subcomputations (frames) which have to be performed before the machine terminates
- Initial and final states
  - initial states: closed expression and an empty stack
  - final states: expression is a value and the stack is empty

Utrecht University

The C-machine

- Example: addition in three stages

1. Evaluate first argument

   - first argument becomes current expression

   - remember to continue with computation, result as first argument

2. Evaluate second argument

   - second argument becomes current expression

   - remember to continue with computation, result as second argument

3. Perform addition

Utrecht University

The C-machine

- How can we denote a stack frame as a term?
- We use terms with holes; e.g.,

$$\texttt{(Plus } \square \texttt{ } e_2 \texttt{)}$$

  - suspended computation of addition
  - waits for the value of its first argument

- Inductive definition of frames:

$$\frac{e \ \mathit{expr}}{\texttt{(Plus } \square \texttt{ } e\texttt{)} \ \mathit{frame}}$$

$$\frac{v \ \mathit{value}}{\texttt{(Plus } v \texttt{ } \square\texttt{)} \ \mathit{frame}}$$

(Plus e_1 □) not a frame, because first argument is evaluated first!

Utrecht University

The C-machine

- Alternatively, we could have different operators for the three different variants of plus frames:

$$\frac{e\ \mathit{expr}}{\texttt{(Plus}\ \square\ e\texttt{)}\ \mathit{frame}}$$

$$\frac{v\ \mathit{value}}{\texttt{(Plus}\ v\ \ \square\texttt{)}\ \ \mathit{frame}}$$

$$\frac{e\ \mathit{expr}}{\texttt{(Plus1}\ e\texttt{)}\ \mathit{frame}}$$

$$\frac{v\ \mathit{value}}{\texttt{(Plus2}\ v\texttt{)}\ \mathit{frame}}$$

- The first representation makes it easier to see what is missing, but the two representations are equivalent

Inductive Definition of Frames

- Addition

$$\frac{e\ \mathit{expr}}{(\texttt{Plus}\ \square\ e)\ \mathit{frame}}$$

$$\frac{v\ \mathit{value}}{(\texttt{Plus}\ v\ \square)\ \mathit{frame}}$$

- If-expressions

$$\frac{e_1\ \mathit{expr} \qquad e_2\ \mathit{expr}}{(\texttt{If}\ \square\ e_1\ e_2)\ \mathit{frame}}$$

- Application (same as addition, just replace operator)
- No frames for `Recfun` (they are expressions/values)

Utrecht University

Example of frames

```
(Plus □ (Num 3))

(Plus □ (Plus (Num 2) (Num 3)))

(Plus □ (If (Const False) (Num 2) (Num 3)))

(Plus (Num 2) □)

(Plus (Plus (Num 4) (Num 2)) □)

(If □ (Num 2) (Num 3))

(Apply (Recfun Int Int (x. Plus x x)) □)

(Apply □ (Plus (Num 3) (Num 4)))
```


Utrecht University

Stack and Machine Modes

- Stacks: $f_1 \triangleright f_2 \triangleright \circ$
  - f_1 *is the top-most frame*
  - f_2 *is the second frame*
  - $\circ$ *is the empty stack*
- Inductive definition:

$$\frac{}{\circ \quad stack}$$

$$\frac{f\ frame \quad s\ stack}{f \triangleright s\ stack}$$

- Machine modes: the C-machine operates in two modes:
  - $s > e$: evaluate expression e under stack s
  - $s < v$: return value v to stack s

Transition Rules for MinHs

- Values (integers, booleans, functions)

$$\frac{}{s \succ v \mapsto_c s \prec v}$$

evaluate the value v under stack s

return the value v to the stack s

- Addition

$$\frac{}{s \succ (\texttt{Plus}\ e_1\ e_2) \mapsto_c (\texttt{Plus}\ \square\ e_2) \triangleright s \succ e_1}$$

$$\frac{}{(\texttt{Plus}\ \square\ e_2) \triangleright s \prec v \mapsto_c (\texttt{Plus}\ v\ \square) \triangleright s \succ e_2}$$

$$\frac{}{(\texttt{Plus}(\texttt{Num}\ n_1)\ \square) \triangleright s \prec (\texttt{Num}\ n_2) \mapsto_c s \prec (\texttt{Num}\ (n_1 + n_2))}$$

Utrecht University

Transition Rules for MinHs

- if-expressions

$$\frac{}{s \succ (\texttt{If}\ e_1\ e_2\ e_3) \mapsto_c (\texttt{If}\ \square\ e_2\ e_3) \triangleright s \succ e_1}$$

$$\frac{}{(\texttt{If}\ \square\ e_2\ e_3) \triangleright s \prec \texttt{True} \mapsto_c s \succ e_2}$$

$$\frac{}{(\texttt{If}\ \square\ e_2\ e_3) \triangleright s \prec \texttt{False} \mapsto_c s \succ e_3}$$

Utrecht University

Transition Rules for MinHs

- Function application

$$s \succ (\texttt{Apply}\ e_1\ e_2) \quad \mapsto_c \quad (\texttt{Apply}\ \square\ e_2) \triangleright s \succ e_1$$

$$(\texttt{Apply}\ \square\ e_2) \triangleright s \prec v \quad \mapsto_c \quad (\texttt{Apply}\ v\ \square) \triangleright s \succ e_2$$

$$(\texttt{Apply}\ \langle f.x.e \rangle)\ \square) \triangleright s \prec v \quad \mapsto_c \quad s \succ e\ [f := \langle f.x.e \rangle, x := v]$$

- Observations;
  - all the inference rules are axioms!
  - the definition of single-step evaluation in the C-machine is not recursive
  - the full evaluator is tail recursive (can be implemented using a while-loop)

from now on to save some space, we use a more compact notation for functions and drop the type arguments:
$\langle f.x.e \rangle$ instead of $(\texttt{Recfun}\ \tau_1\ \tau_2\ (f.x.e))$

Environments

- Now, let's get rid of substitution!
- Let's first look at a big step environment semantics

$$\frac{e_1 \Downarrow \langle f.x.e \rangle \qquad e_2 \Downarrow v \qquad e[f := \langle f.x.e \rangle, x := v] \Downarrow v'}{\texttt{(Apply } e_1\ e_2\texttt{)} \Downarrow v'}$$

- Without substitution, we need an environment η for the evaluation
  - An environment η as is an ordered (possibly empty) sequence of bindings
  - Lookup (retrieves left-most entry): $\eta(x) = v$

$$\frac{}{\bullet\ env} \qquad \frac{\eta\ env}{x = v,\ \eta\ env}$$

- Evaluation of expression e under environment η :

$$\eta \mid e \Downarrow v$$

Environments

- Big-step environment rule for application:

$$\frac{\eta(x) = v}{\eta \mid x \Downarrow v} \qquad \frac{}{\eta \mid f.x.e \Downarrow \langle f.x.e \rangle}$$

$$\frac{\eta \mid e_1 \Downarrow \langle f.x.e \rangle \qquad \eta \mid e_2 \Downarrow v \qquad f = \langle f.x.e \rangle, x = v, \eta \mid e \Downarrow v'}{\eta \mid (\texttt{Apply}\ e_1\ e_2) \Downarrow v'}$$

Utrecht University

Environments

- Does this work?

```
(recfun addOne :: x =  x + 1)) 15
```

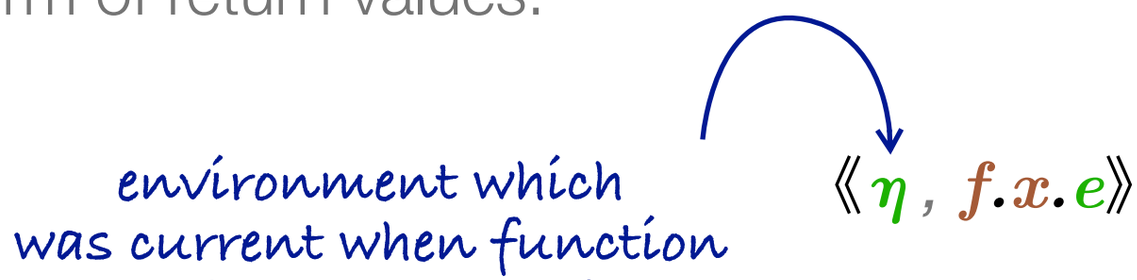
```
(recfun add :: x =
   recfun add' :: y = x + y))    15  5
```

Environments

- Problem:
  - functions as return values may contain free variables, which escape their scopes
  - when the body is finally evaluated, the information what the free variables were bound to is not available anymore

Dealing with partial application

- Solution:
  - we need to bundle returned functions with current environment
  - we call this a closure
  - requires a new form of return values:



  - Closures only appear as values during execution - there is no source form

Environments

- Big-step semantics rules for function values and application:

$$\frac{}{\eta \mid \langle f.x.e \rangle \Downarrow \langle\!\langle f.x.e, \eta \rangle\!\rangle}$$

$$\frac{\eta \mid e_1 \Downarrow \langle\!\langle f.x.e, \eta' \rangle\!\rangle \qquad \eta \mid e_2 \Downarrow v \qquad f = \langle\!\langle f.x.e, \eta' \rangle\!\rangle \,, x = v, \eta' \mid e \Downarrow v'}{\eta \mid (\texttt{Apply}\ e_1\ e_2) \Downarrow v'}$$

Environments

- Closures are necessary whenever we have functions as return values, which contain variables bound outside that function
- Example: JavaScript

```
function createCounter() {
    let count = 1; // This is the variable that will be remembered by the closure.

    return function() { // This returned function forms a closure.
        count++; // It accesses the `count` variable from its lexical scope.
        return count;
    };
}

const counter1 = createCounter(); // Create a counter instance
console.log(counter1());
console.log(counter1());
console.log(counter1());

const counter2 = createCounter();
console.log(counter2());
console.log(counter2());
console.log(counter1());
```

Environments

- Let's get back to our C-machine and add environments:

$$(\texttt{Apply}\ \langle f.x.e\rangle\ \square) \triangleright s \prec v \quad \mapsto_c \quad s \succ e\ [f := \langle f.x.e\rangle, x := v]$$

- We also need an environment η as part of the state

$$s \mid \eta \succ e$$

$$s \mid \eta \prec v$$

Environments

- Just like in the big step semantics, we need to have a special rule for returning function values:

$$s \mid \eta \succ (\texttt{Num n}) \quad \mapsto_E \quad s \mid \eta \prec (\texttt{Num n})$$

$$s \mid \eta \succ \langle f.x.e \rangle \quad \mapsto_E \quad s \mid \eta \prec \langle\!\langle \eta, f.x.e \rangle\!\rangle$$

- How does application work?

$$(\texttt{Apply } \langle\!\langle \eta', f.x.e \rangle\!\rangle \ \square) \triangleright s \mid \eta \prec v \mapsto_E \ ?$$

Utrecht University

Environments

- We need to save the old environment somewhere, and restore it when returning from the function call

  - in TinyC, we discarded the local decs used in the premise:

$$\frac{(g.l,\ ss) \Downarrow (g'.l',\ rv;)}{(g,\ \{l\ ss\}) \Downarrow (g',\ rv;)}$$

- Let's use the stack to store the old environment!

The E-machine

- In the E-machine
  - we have frames defined exactly as before
  - explicit environments, which are a **ordered** sequence of variable bindings

$$\frac{}{\bullet \quad env} \qquad \frac{\eta \;\; env}{x = v,\ \eta \;\; env}$$

  - stacks in the E-machine are sequences of environments and frames

$$\frac{}{\circ \quad stack} \qquad \frac{f \; frame \quad s \; stack}{f \triangleright s \;\; stack} \qquad \frac{\eta \; env \qquad s \;\; stack}{\eta \triangleright s \;\; stack}$$

  - states in the E-machine include an environment

$$s \mid \eta \succ e$$

$$s \mid \eta \prec v$$

Utrecht University

The E-machine: Transition Rules

- Variables:

$$s \mid \eta \succ x \quad \mapsto_E \quad s \mid \eta \prec v \quad \text{, if } \eta(x) = v$$

- Application:

$$\texttt{(Apply}\ \langle\!\langle \eta', f.x.e \rangle\!\rangle\ \texttt{)}\ \square) \triangleright s \mid \eta \prec v \quad \mapsto_E \quad \eta \triangleright s \mid (f = \langle\!\langle \eta', f.x.e \rangle\!\rangle, x = v, \eta') \succ e$$

restore environment from closure, add binding for argument x and function f

- Returning from a function call:

$$\eta \triangleright s \mid \eta' \prec v \quad \mapsto_E \quad s \mid \eta \prec v$$