Programming Language Concepts, CS2104 Lecture 3

Statements, Kernel Language, Abstract Machine

Reminder of last lecture

- Programming language definition: syntax, semantics
  - CFG, EBNF
- Data structures
  - simple: integers, floats, literals
  - compound: records, tuples, lists
- Kernel language
  - linguistic abstraction
  - data types
  - variables and partial values
  - unification

Overview

- Some Oz concepts
  - Pattern matching
  - Tail recursion
  - Lazy evaluation
- Kernel language
  - statements and expressions
- Kernel language semantics
  - Use *operational semantics*
    - Aid programmer in reasoning and understanding
  - The model is a sort of an *abstract machine*, but leaves out details about registers and explicit memory address
    - Aid implementer to do an efficient execution on a real machine

Pattern-Matching on Numbers

```
fun {Fact N}
   case N
   of 0 then 1
   [] N then N*{Fact (N-1)} end
end
```

Pattern Matching on Structures

```
fun {Depth T}
   case Xs of
      leaf(value:_)  then 1
   [] node(left:L right:R value:_)
           then 1+{Max {Depth L} {Depth R}}
   end
end
```


Compared to Conditional

```
fun {SumList Xs}
   case Xs
   of nil  then 0
   [] X|Xr then X + {SumList Xr} end
end
```

Using only Conditional

```
fun {SumList Xs}
    if {Label Xs}=='nil' then 0
    elseif {Label Xs}=='|' andthen {Width Xs}==2
              then Xs.1 +{SumList Xs.2}
    end
end
```


Linear Recursion

```
fun {Fact N}
   case N
   of 0  then 1
   [] N then N * {Fact (N-1)} end
end
```

```
{Fact 3}
⇒ 3*{Fact 2}
⇒ 3*(2*{Fact 1})
⇒ 3*(2*(1*{Fact 0}))
⇒ 3*(2*(1*1))
⇒ 3*(2*1)
⇒ 3*2
⇒ 6
```

going down recursion

return from recursion

Accumulating Parameter

```
fun {Fact N } {FactT N 1} end
```

Accumulating Parameter

```
fun {FactT N Acc}
   case N
   of 0  then Acc
   [] N then {FactT (N-1) N*Acc} end
end
```

Accumulating Parameter

```
{Fact 3}
⇒ {FactT 3 1}
⇒ {FactT 2 3*1}
⇒ {FactT 2 3})
⇒ {FactT 1 2*3}
⇒ {FactT 1 6}
⇒ {FactT 0 1*6}
⇒ {FactT 0 6}
⇒ 6
```

going down recursion and accumulating result in parameter

Accumulating Parameter = Tail Recursion = Loop!

Tail Recursion = Loop

```
fun {FactT N Acc}
   case N
   of 0  then Acc
   [] N then N=N-1
             Acc=N*Acc
             {FactT N Acc}
   end
end
```

jump

Last call = Tail call

Lazy Evaluation

Infinite list of numbers!

```
fun lazy {Ints N} N|{Ints N+1} end
```

```
{Ints 2}
⇒ 2|{Ints 3}
⇒ 2|(3|{Ints 4})
⇒ 2|(3|(4|{Ints 5}))
⇒ 2|(3|(4|(5|{Ints 6})))
⇒ 2|(3|(4|(5|(6|{Ints 7}))))
      :
```

What if we were to compute : `{SumList {Ints 2}}` ?

Taking first N elements of List

```
fun {Take L N}
  if N<=0 then nil
  else case L of
          nil then nil
       [] X|Xs then X|{Take Xs (N-1)} end end
end
```

```
{Take [a b c d] 2}
⇒ a|{Take [b c d] 1}
⇒ a|b|{Take [c d] 0}
⇒ a|b|nil

{Take {Ints 2} 2}
⇒  ?
```

Eager Evaluation

```
{Take {Ints 2} 2}
⇒ {Take 2|{Ints 3} 2}
⇒ {Take 2|(3|{Ints 4}) 2}
⇒ {Take 2|(3|(4|{Ints 5}))} 2}
⇒ {Take 2|(3|(4|(5|{Ints 6}))) 2}
⇒ {Take 2|(3|(4|(5|(6|{Ints 7})))) 2}
        :
```

Loop as Infinite list eagerly evaluated!

Lazy Evaluation

Evaluate the lazy argument only as needed

```
{Take {Ints 2} 2}
⇒ {Take 2|{Ints 3} 2}
⇒ 2|{Take {Ints 3} 1}
⇒ 2|{Take 3|{Ints 4} 1}
⇒ 2|(3|{Take {Ints 4} 0})
⇒ 2|(3|nil)
```

terminates despite infinite list

Kernel Concepts

- Single-assignment store
- Environment
- Semantic statement
- Execution state and Computation
- Statements Execution for:
  - skip and sequential composition
  - variable declaration
  - store manipulation
  - conditional

Procedure Declarations

- Kernel language

  $\langle x\rangle$ = `proc {$` $\langle y_1\rangle \ldots \langle y_n\rangle$`}` $\langle s\rangle$ `end`

  is a legal statement
  - binds $\langle x\rangle$ to procedure value
  - declares (introduces a procedure)
- Familiar syntactic variant

  `proc {`$\langle x\rangle$ $\langle y_1\rangle \ldots \langle y_n\rangle$`}` $\langle s\rangle$ `end`

  introduces (declares) the procedure $\langle x\rangle$
- A procedure declaration is a value, whereas a procedure application is a statement!

What Is a Procedure?

- It is a **value** of the **procedure type**.
  - Java: methods with `void` as return type

```
declare
X = proc {$ Y}          →  $ is the nesting operator
      {Browse 2*Y}
    end
{X 3}            →  6
{Browse X}       →  <P/1 X>
```

- But how to return a result (as parameter) anyway?
  - Idea: use an unbound variable
  - Why: we can supply its value after we have computed it!

Operations on Procedures

- Three basic operations:
  - Defining them (with `proc` statement)
  - Calling them (with `{ }` notation)
  - Testing if a value is a procedure
    - `{IsProcedure P}` returns `true` if `P` is a procedure, and `false` otherwise

```
declare
X = proc {$ Y}
      {Browse 2*Y}
    end
{Browse {IsProcedure X}}
```


Towards Computation Model

- Step One: Make the language small
  - Transform the language of function on partial values to a small kernel language
- Kernel language

procedures	no functions
records	no tuple syntax
	no list syntax
local declarations	no nested calls
	no nested constructions

From Function to Procedure

```
fun {Sum Xs}
   case Xs
   of nil then 0
   [] X|Xr then X+{Sum Xr}
   end
end
```

- Introduce an output parameter for procedure

```
proc {SumP Xs N}
   case Xs
   of nil then N=0
   [] X|Xr then N=X+{Sum Xr}
   end
end
```

Why we need `local` statements?

```
proc {SumP Xs N}
   case Xs
   of nil then N=0
   [] X|Xr then
      local M in {SumP Xr M} N=X+M end
   end
end
```

- Local declaration of variables supported.
- Needed to allow kernel language to be based entirely on procedures

How `N` was actually transmitted?

- Having the call `{SumP [1 2 3] C}`, the identifier `Xs` is bound to `[1 2 3]` and `C` is unbound.
- At the callee of `SumP`, whenever `N` is being bound, so will be `C`.
- This way of passing parameters is called **call by reference**.
- Procedures output are passed as references to unbound variables, which are bound inside the procedure.

Local Declarations

```
local X in … end
```

- Introduces the variable identifier `X`
  - visible between `in` and `end`
  - called scope of the variable/declaration
- Creates a new store variable
- Links environment identifier to store variable

Abbreviations for Declarations

- Kernel language
  - just one variable introduced at a time
  - no assignment when first declared
- Oz language syntax supports:
  - several variables at a time
  - variables can be also assigned (initialized) when introduced

Transforming Declarations Multiple Variables

```
local X Y in
   ⟨statement⟩
end
```

⇨

```
local X in
   local Y in
      ⟨statement⟩
   end
end
```


Transforming away Declarations' Initialization

```
local
   X=⟨expression⟩
in
   ⟨statement⟩
end
```

⇨

```
local X in
   X=⟨expression⟩
   ⟨statement⟩
end
```


Transforming Expressions

- Replace function calls by procedure calls
- Use local declaration for intermediate values
- Order of replacements:
  - left to right
  - innermost first
  - it is different for record construction: outermost first
  - Left associativity: `1+2+3` means `((1+2)+3)`
  - Right associativity: `a|b|X` means `(a|(b|X))`, so build the first `'|'`, then the second `'|'`

Function Call to Procedure Call

```
X={F Y}
```

⇨

```
{F Y X}
```

Replacing Nested Calls

```
{P {F X Y} Z}
```

⇨

```
local U1 in
   {F X Y U1}
   {P U1 Z}
end
```


Replacing Nested Calls

```
{P {F {G X} Y} Z}
```

⇨

```
local U2 in
   local U1 in
      {G X U1}
      {F U1 Y U2}
   end
   {P U2 Z}
end
```


Replacing Conditionals

```
if X>Y then
   …
else
   …
end
```

⇨

```
local B in
   B = (X>Y)
   if B then
      …
   else
      …
   end
end
```


Expressions to Statements

```
X = if B then
       …
    else
       …
    end
```

⇨

```
if B then
   X = …
else
   X = …
end
```

Functions to Procedures: Length (0)

```
fun {Length Xs}
   case Xs
   of nil then 0
   [] X|Xr then 1+{Length Xr}
   end
end
```


Functions to Procedures: Length (1)

```
proc {Length Xs N}
   N=case Xs
     of nil then 0
     [] X|Xr then 1+{Length Xr}
     end
end
```

- Make it a procedure

Functions to Procedures: Length (2)

```
proc {Length Xs N}
   case Xs
   of nil then N=0
   [] X|Xr then N=1+{Length Xr}
   end
end
```

- Expressions to statements

Functions to Procedures: Length (3)

```
proc {Length Xs N}
   case Xs
   of nil then N=0
   [] X|Xr then
      local U in
         {Length Xr U}
         N=1+U
      end
   end
end
```

- Replace function call by its corresponding proc call.

Functions to Procedures: Length (4)

```
proc {Length Xs N}
   case Xs
   of nil then N=0
   [] X|Xr then
      local U in
         {Length Xr U}
         {Number.'+' 1 U N}
      end
   end
end
```

- Replace operation (+, dot-access, <, >, …): procedure!

Kernel Language Statement Syntax

⟨s⟩ denotes a statement

⟨s⟩ ::= skip	*empty statement*
| ⟨x⟩ = ⟨y⟩	*variable-variable binding*
| ⟨x⟩ = ⟨v⟩	*variable-value binding*
| ⟨s_1⟩ ⟨s_2⟩	*sequential composition*
| local ⟨x⟩ in ⟨s_1⟩ end	*declaration*
| if ⟨x⟩ then ⟨s_1⟩ else ⟨s_2⟩ end	*conditional*
| { ⟨x⟩ ⟨y_1⟩ … ⟨y_n⟩ }	*procedure application*
| case ⟨x⟩ of ⟨pattern⟩ then ⟨s_1⟩ else ⟨s_2⟩ end	*pattern matching*
⟨v⟩ ::= …	*value expression*
⟨pattern⟩ ::= …	

Abstract Machine

- *Environment* maps variable identifiers to store entities
- *Semantic statement* is a pair of:
  - statement
  - environment
- *Execution state* is a pair of:
  - stack of semantic statements
  - single assignment store
- *Computation* is a sequence of execution states
- An **abstract machine** performs a computation

Single Assignment Store

- Single assignment store σ
  - set of store variables
  - partitioned into
    - sets of variables that are equivalent but unbound
    - variables bound to a value (number, record or procedure)
- Example store $\{x_1, x_2{=}x_3, x_4{=}\mathrm{a}|x_2\}$
  - x_1 unbound
  - x_2, x_3 equal and unbound
  - x_4 bound to partial value a|x_2

Environment

- Environment ***E***
  - maps variable identifiers to entities in store σ
  - written as set of pairs X → *x*
    - identifier X
    - store variable *x*
- Example of environment: { X → *x*, Y → *y* }
  - maps identifier X to store variable *x*
  - maps identifier Y to store variable *y*

Environment and Store

- Given: environment *E*, store σ
- Looking up value for identifier X:
  - find store variable in environment using *E*(X)
  - take value from σ for *E*(X)
- Example:

  $\sigma=\{x_1, x_2=x_3, x_4=\mathrm{a}|x_2\}$ E = { X → x_1, Y → x_4 }
  - E(X) = x_1 where no information in σ on x_1
  - E(Y) = x_4 where σ binds x_4 to a|x_2

Calculating with Environments

- Program execution looks up values
  - assume store σ
  - given identifier ⟨x⟩
  - E(⟨x⟩) is the value of ⟨x⟩ in store σ
- Program execution modifies environments
  - for example: declaration
  - add mappings for new identifiers
  - overwrite existing mappings
  - restrict mappings on sets of identifiers

Environment Adjunction

- Given: Environment *E*

then $E + \{\langle x\rangle_1 \to x_1, \ldots, \langle x\rangle_n \to x_n\}$

is a new environment *E*' with mappings added:
  - always take store entity from new mappings
  - might overwrite (or shadow) old mappings

Environment Projection

- Given: Environment E

  $E \mid \{\langle x\rangle_1, \ldots, \langle x\rangle_n\}$

  is a new environment E' where only mappings for $\{\langle x\rangle_1, \ldots, \langle x\rangle_n\}$ are retained from E

Adjunction Example

- $E_0 = \{\langle Y\rangle \rightarrow 1\}$
- $E_1 = E_0 + \{\langle X\rangle \rightarrow 2\}$
  - corresponds to $\{\langle X\rangle \rightarrow 2, \langle Y\rangle \rightarrow 1\}$
  - $E_1(\langle X\rangle) = 2$
- $E_2 = E_1 + \{\langle X\rangle \rightarrow 3\}$
  - corresponds to $\{\langle X\rangle \rightarrow 3, \langle Y\rangle \rightarrow 1\}$
  - $E_2(\langle X\rangle) = 3$

Why Adjunction?

E_0

```
local X in
   X = 2
   local X in
      X = 3        } E2
      {… Y}
   end
   …               } E1
end
```

E_0

Semantic Statements

- Semantic statement **(⟨s⟩, *E*)**
  - pair of (statement, environment)
- To actually execute statement:
  - environment to map identifiers
    - modified with execution of each statement
    - each statement has its own environment
  - store to find values
    - all statements modify same store
    - single store

Stacks of Statements

- Execution maintains stack of semantic statements **ST =** $[(\langle s\rangle_1, E_1), \ldots, (\langle s\rangle_n, E_n)]$
  - always topmost statement $(\langle s\rangle_1, E_1)$ executes first
    - <s> is statement
    - E denotes the environment mapping
  - rest of stack: remaining work to be done
- Also called: *semantic stack*

Execution State

- *Execution state* **(ST, σ)**
  - pair of (semantic stack, store)
- *Computation*

  $(ST_1, \sigma_1) \Rightarrow (ST_2, \sigma_2) \Rightarrow (ST_3, \sigma_3) \Rightarrow \ldots$
  - sequence of execution states

Program Execution

- Initial execution state

  $([(\langle s\rangle,\varnothing)] , \varnothing)$
  - empty store $\varnothing$
  - stack with semantic statement $[(\langle s\rangle,\varnothing)]$
    - single statement $\langle s\rangle$, empty environment $\varnothing$
- At each execution step
  - pop topmost element of semantic stack
  - execute according to statement
- If semantic stack is empty, then execution stops

Semantic Stack States

- Semantic stack can be in following states
  - *terminated* stack is empty
  - *runnable* can do execution step
  - *suspended* stack not empty, no execution step possible
- Statements
  - *non-suspending* can always execute
  - *suspending* need values from store dataflow behavior

Summary up to now

- Single assignment store — σ
- Environments — *E*
  - adjunction, projection — *E* + {…}   *E* | $_{\{\ldots\}}$
- Semantic statements — (⟨s⟩, *E*)
- Semantic stacks — [(⟨s⟩, *E*) …]
- Execution state — (*ST*, σ)
- Computation = sequence of execution states
- Program execution
  - runnable, terminated, suspended
- Statements
  - suspending, non-suspending

Statement Execution

- Simple statements
  - skip and sequential composition
  - variable declaration
  - store manipulation
  - Conditional (`if` statement)
- Computing with procedures (later lecture)
  - lexical scoping
  - closures
  - procedures as values
  - procedure call

Simple Statements

⟨s⟩ denotes a statement

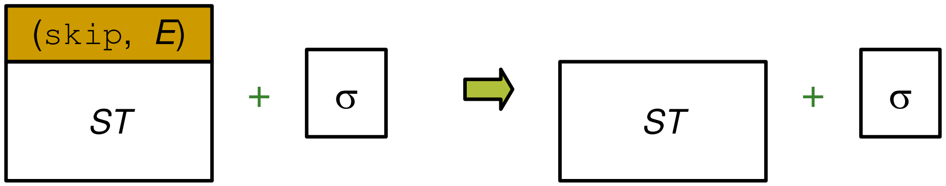
⟨s⟩ ::=	skip		*empty statement*
			⟨x⟩ = ⟨y⟩	*variable-variable binding*
			⟨x⟩ = ⟨v⟩	*variable-value binding*
			$\langle s_1\rangle\ \langle s_2\rangle$	*sequential composition*
			local ⟨x⟩ in $\langle s_1\rangle$ end	*declaration*
			if ⟨x⟩ then $\langle s_1\rangle$ else $\langle s_2\rangle$ end	*conditional*
⟨v⟩ ::=	…		*value expression* (no procedures here)

Executing `skip`

- Execution of semantic statement

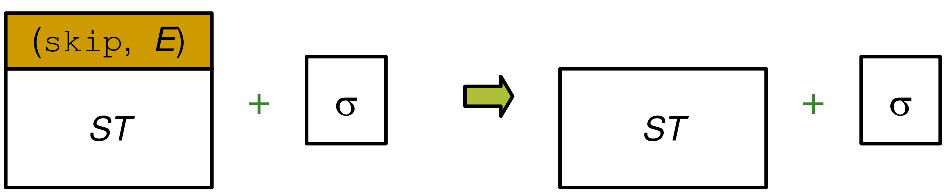
  (`skip`, *E*)
- Do nothing
  - means: continue with next statement
  - non-suspending statement

Executing `skip`



- No effect on store σ
- Non-suspending statement

Executing `skip`

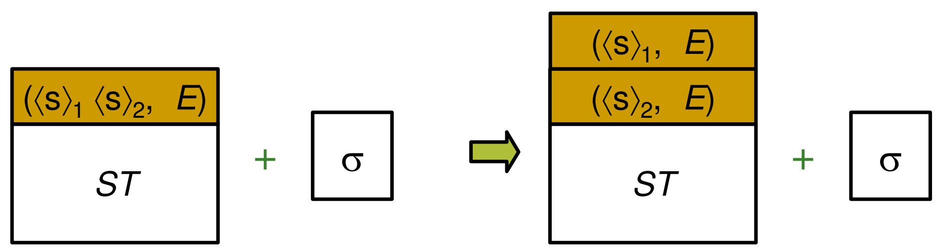


- Remember: topmost statement is always popped!

Executing Sequential Composition

- Semantic statement is
  $(\langle s\rangle_1\ \langle s\rangle_2,\ E)$
- Push in following order
  - $\langle s\rangle_2$ executes after
  - $\langle s\rangle_1$ executes next
- Statement is non-suspending

Sequential Composition

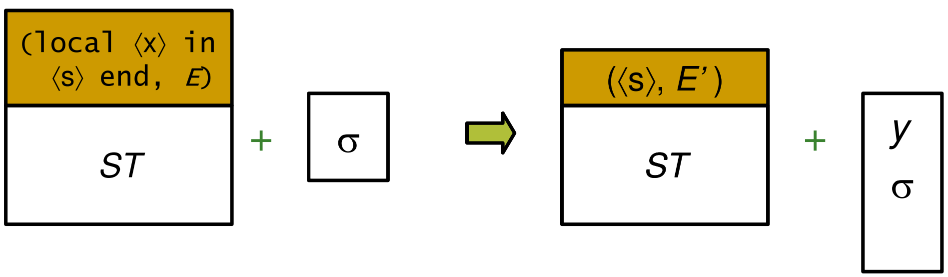


- Decompose statement sequences
  - environment is given to both statements

Executing `local`

- Semantic statement is
  (`local` ⟨x⟩ `in` ⟨s⟩ `end`, *E*)
- Execute as follows:
  - create new variable *y* in store
  - create new environment *E'* = *E* + {⟨x⟩ → *y*}
  - push (⟨s⟩, *E'*)
- Statement is non-suspending

Executing `local`

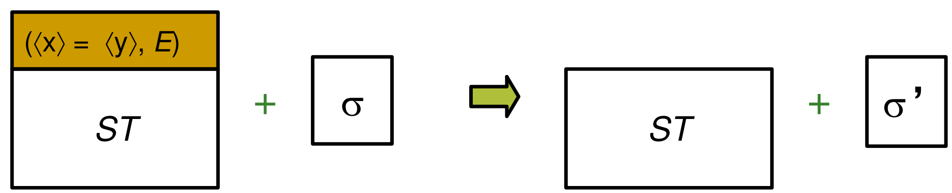


- With *E'* = *E* + {⟨x⟩ → *y*}

Variable-Variable Equality

- Semantic statement is
  (⟨x⟩ = ⟨y⟩, *E*)
- Execute as follows
  - bind *E*(⟨x⟩) and *E*(⟨y⟩) in store
- Statement is non-suspending

Executing Variable-Variable Equality



- σ' is obtained from σ by binding *E*(⟨x⟩) and *E*(⟨y⟩) in store

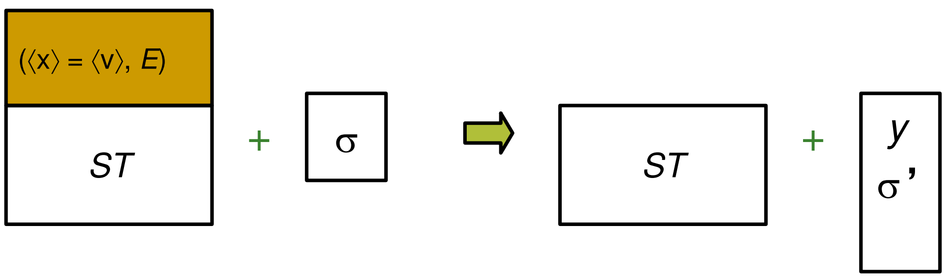
Variable-Value Equality

- Semantic statement is

  (⟨x⟩ = ⟨v⟩, *E*)

  where ⟨v⟩ is a number or a record (procedures will be discussed later)
- Execute as follows
  - create a variable y in store and let y refers to value ⟨v⟩
  - any identifier ⟨z⟩ from ⟨v⟩ is replaced by *E*(⟨z⟩)
  - bind *E*(⟨x⟩) and y in store
- Statement is non-suspending

Executing Variable-Value Equality



- y refers to value ⟨v⟩
- Store σ is modified into σ’ such that:
  - any identifier ⟨z⟩ from ⟨v⟩ is replaced by *E*(⟨z⟩)
  - bind *E*(⟨x⟩) and y in store σ

Suspending Statements

- All statements so far can always execute
  - non-suspending (or immediate)
- Conditional?
  - requires condition ⟨x⟩ to be bound variable
  - *activation condition*: ⟨x⟩ is bound (determined)

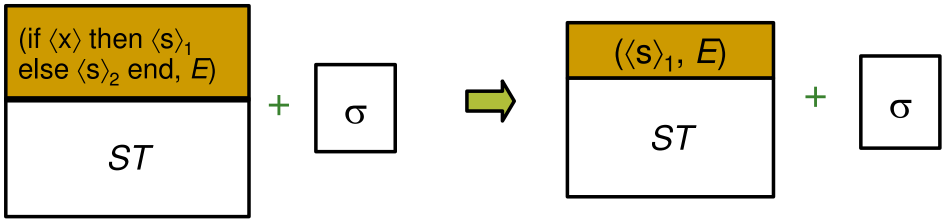
Executing `if`

- Semantic statement is

  (`if` ⟨x⟩ `then` $\langle s \rangle_1$ `else` $\langle s \rangle_2$ `end`, *E*)
- If the activation condition “bound(⟨x⟩)” is `true`
  - if *E*(⟨x⟩) bound to `true` push $\langle s \rangle_1$
  - if *E*(⟨x⟩) bound to `false` push $\langle s \rangle_2$
  - otherwise, raise error
- Otherwise, suspend the `if` statement…

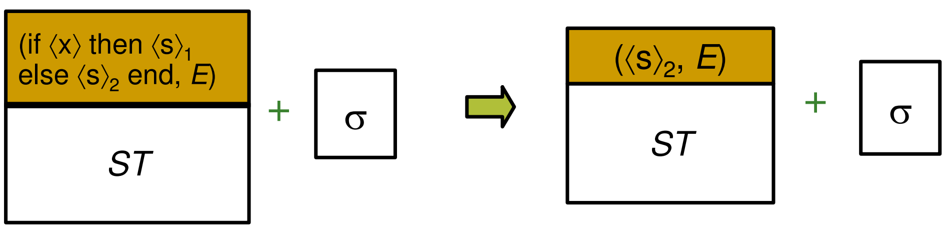
Executing `if`

- If the activation condition "bound(⟨x⟩)" is `true`
  - if *E*(⟨x⟩) bound to `true`



Executing `if`

- If the activation condition "bound(⟨x⟩)" is `true`
  - if *E*(x⟩) bound to `false`



An Example

```
local X in
   local B in
      B=true
      if B then X=1 else skip end
   end
end
```

- We can reason that `X` will be bound to `1`

Example: Initial State

```
([(local X in
      local B in
         B=true
         if B then X=1 else skip end
      end
   end, ∅)],
 ∅)
```

- Start with empty store and empty environment

Example: `local`

```
([(local B in
     B=true
     if B then X=1 else skip end
   end,
   {X → x})],
 {x})
```

- Create new store variable *x*
- Continue with new environment

Example: `local`

```
([(B=true
   if B then X=1 else skip end
   ,
   {B → b, X → x})],
 {b,x})
```

- Create new store variable *b*
- Continue with new environment

Example: Sequential Composition

```
([(B=true, {B → b, X → x}),
  (if B then X=1
   else skip end, {B → b, X → x})],
 {b,x})
```

- Decompose to two statements
- Stack has now two semantic statements

Example: Variable-Value Assignment

```
([(if B then X=1
   else skip end, {B → b, X → x})],
 {b=true, x})
```

- Environment maps `B` to *b*
- Bind *b* to `true`

Example: `if`

([(`X=1`, {`B` → *b*, `X` → *x*})],
{*b*=`true`, *x*})

- Environment maps `B` to *b*
- Bind *b* to `true`
- Because the activation condition "bound(⟨x⟩)" is `true`, continue with `then` branch of `if` statement

Example: Variable-Value Assignment

([],
{*b*=`true`, *x*=`1`})

- Environment maps `X` to *x*
- Binds *x* to `1`
- Computation terminates as stack is empty

Summary up to now

- Semantic statement execute by
  - popping itself — always
  - creating environment — `local`
  - manipulating store — `local`, `=`
  - pushing new statements — `local`, `if`, sequential composition
- Semantic statement can suspend
  - activation condition (`if` statement)
  - read store

Pattern Matching

- Semantic statement is

  (**case** ⟨x⟩
  of ⟨lit⟩(⟨feat⟩$_1$:⟨y⟩$_1$ … ⟨feat⟩$_n$:⟨y⟩$_n$) **then** ⟨s⟩$_1$
  else ⟨s⟩$_2$ **end**, *E*)
- It is a suspending statement
- Activation condition is: "bound(⟨x⟩)"
- If activation condition is `false`, then suspend!

Pattern Matching

- Semantic statement is
  (**case** $\langle x\rangle$
  of $\langle lit\rangle(\langle feat\rangle_1{:}\langle y\rangle_1 \ldots \langle feat\rangle_n{:}\langle y\rangle_n)$ **then** $\langle s\rangle_1$
  else $\langle s\rangle_2$ **end**, E)
- If $E(\langle x\rangle)$ matches the pattern, that is,
  - label of $E(\langle x\rangle)$ is $\langle lit\rangle$ and
  - its arity is $[\langle feat\rangle_1 \ldots \langle feat\rangle_n]$,
- then push
  $(\langle s\rangle_1,$
  $E + \{\langle y\rangle_1 \rightarrow E(\langle x\rangle).\langle feat\rangle_1 ,$
  $\ldots ,$
  $\langle y\rangle_n \rightarrow E(\langle x\rangle).\langle feat\rangle_n \})$

Pattern Matching

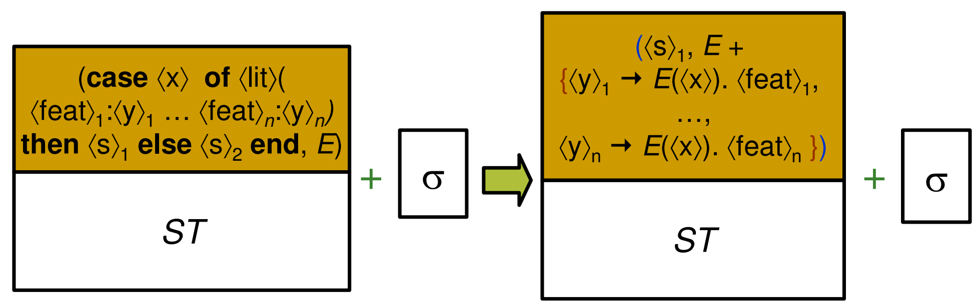
- Semantic statement is
  (**case** $\langle x\rangle$
  of $\langle lit\rangle(\langle feat\rangle_1{:}\langle y\rangle_1 \ldots \langle feat\rangle_n{:}\langle y\rangle_n)$ **then** $\langle s\rangle_1$
  else $\langle s\rangle_2$ **end**, E)
- If $E(\langle x\rangle)$ does not match pattern, push
  $(\langle s\rangle_2, E)$

Pattern Matching

- Semantic statement is
  (**case** $\langle x\rangle$
  of $\langle lit\rangle(\langle feat\rangle_1{:}\langle y\rangle_1 \ldots \langle feat\rangle_n{:}\langle y\rangle_n)$ **then** $\langle s\rangle_1$
  else $\langle s\rangle_2$ **end**, E)
- It does not introduce new variables in the store
- Identifiers $\langle y\rangle_1 \ldots \langle y\rangle_n$ are visible only in $\langle s\rangle_1$

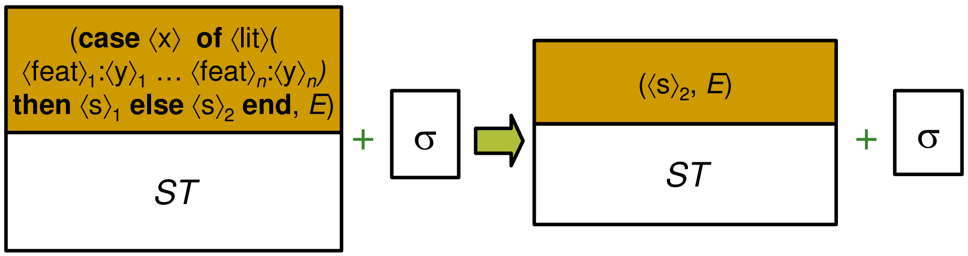
Executing `case`

- If the activation condition "bound($\langle x\rangle$)" is `true`
  - if $E(\langle x\rangle)$ *matches* the pattern



Executing `case`

- If the activation condition "bound(⟨x⟩)" is `true`
  - if *E*(⟨x⟩) *does not* match the pattern



Example: `case` Statement

```
([(case X of
      f(X1 X2) then Y = g(X2 X1)
    else Y = c
    end,
 {X →v1, Y →v2})], % Env
 {v1=f(v3 v4), v2, v3=a, v4=b} % Store
)
```

- We declared `X`, `Y`, `X1`, `X2` as local identifiers and `X=f(v3 v4)`, `X1=a` and `X2=b`
- What is the value of `Y` after executing `case`?

Example: `case` Statement

```
([(Y = g(X2 X1),
  {X →v1, Y →v2, X1 →v3, X2 →v4})
  ],
  {v1=f(v3 v4), v2, v3=a, v4=b}
)
```

- The activation condition "bound(⟨x⟩)" is `true`
- Remember that `X1=a, X2=b`

Example: `case` Statement

```
([],
  {v1=f(v3 v4),
   v2=g(v4 v3),v3=a,v4=b}
)
```

- Remember `Y` refers to v2, so

  `Y = g(b a)`

Summary

- Kernel language
  - linguistic abstraction
  - data types
  - variables and partial values
  - statements and expressions
- Computing with procedures (next lecture)
  - lexical scoping
  - closures
  - procedures as values
  - procedure call

Reading Suggestions

- from [van Roy,Haridi; 2004]
  - Chapter 2, Sections 2.1.1-2.3.5, 2.8
  - Appendices B, C, D
  - Exercises 2.9.1-2.9.3, 2.9.13