Administrative

In Homework 1, problem 2.3 from Mitchell's book should be 2.3.2.

For Thursday, read the Intro to ML paper. Note there is an implementation of ML on the EECS machines in `/usr/sww/bin/sml`.

Semantics Review

Axiomatic semantics give a proof system for proving the equivalence between programs.

+ For simple languages like PCF, axioms are like program execution rules.

- Says nothing about termination.

Operational (and Structure Operational) semantics define an abstract interpreter.

+ Relatively easy to read.

+ Simple proofs.

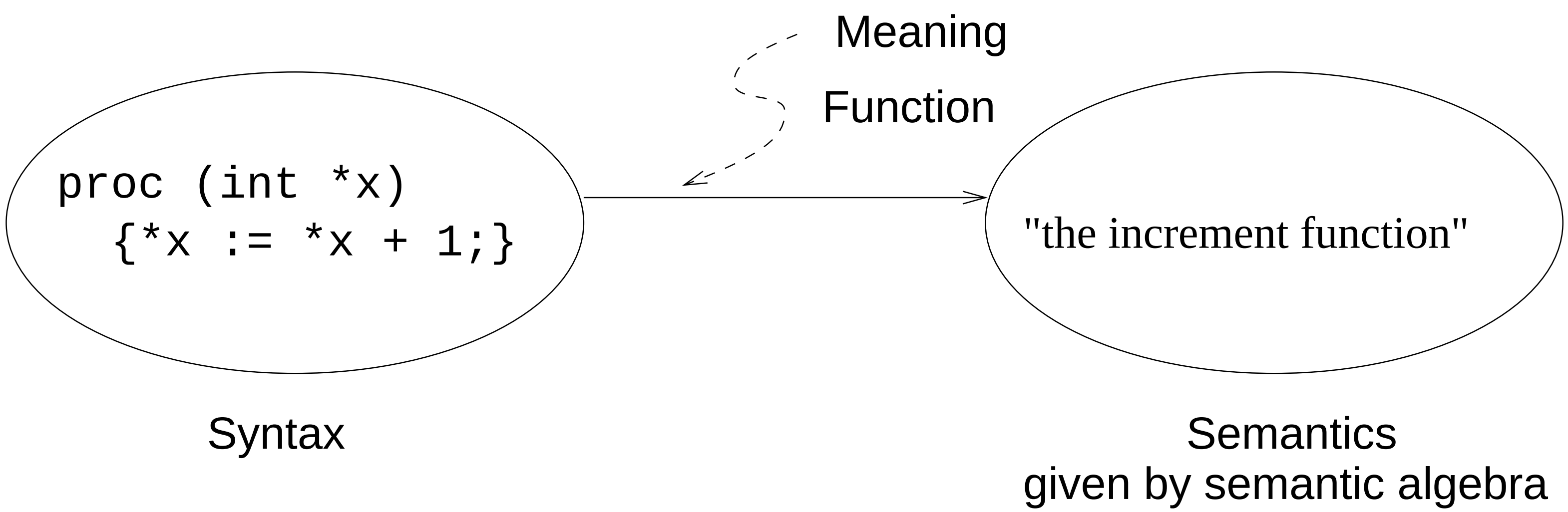
- Overly constrains meaning, implementation bias.

- Not compositional.

- Does not define a mathematical model.

Denotational Semantics

Each type express is associated with a set of values.



An "algebra" is a set of objects and functions on those objects

- A term of type σ is one of the values in this set.
- A lambda term $\lambda x : \sigma . M$ is interpreted as a mathematical function.
- A term of type $\sigma \rightarrow \tau$ is a set of functions.

Note: It is easier to define a denotational semantics for the *typed* lambda calculus, and derive untyped from that.

Denotational Semantics of PCF

Each expression in PCF will be interpreted as either a boolean, natural number, function, or special non-terminating value called *bottom*, $\perp$.

type name	values
$bool$	$true, false, \perp_{bool}$
nat	$0, 1, 2, \ldots \text{or} \perp$
$bool \rightarrow bool$	$true \rightarrow false, \text{etc., and} \perp bool \rightarrow bool$

Some mappings from lambda expression to their meanings:

value name	meaning
$true$	$true$
$false$	$false$
$\lambda\ x : bool\ .\ x$	identity function on bool
$\sigma \times \sigma$	pairs of σ values

The interpretation is defined by a *meaning* or *valuation* function on terms.

A Valuation Function for PCF

A problem with defining the meaning of general lambda terms is that they may have free variables. We can only define the semantics of that term in a given *environment* denoted by η.
The notation used for the meaning of a term M in some environment η is: $[\![M]\!]\eta$
For variable free constants, the environment doesn't matter:

$$\begin{aligned}
[\![1]\!]\eta &= 1 \\
[\![2]\!]\eta &= 2 \\
[\![true]\!]\eta &= \text{true} \\
[\![false]\!]\eta &= \text{false}
\end{aligned}$$

For terms with variables, we need to look at the environment, which maps variables (syntax) to semantic values (e.g., numbers, not numerals).

$$\begin{aligned}
[\![x]\!]\eta &= \eta(x) \\
[\![M\ N]\!]\eta &= [\![M]\!]\eta[\![N]\!]\eta
\end{aligned}$$

Compound Constructs

The conditional construct is defined as:

$$
\begin{aligned}
[\![\texttt{if } B \texttt{ then } M \texttt{ else } N\,]\!]\eta &= [\![M]\!]\eta \text{ if } [\![B]\!]\eta \text{ is true} \\
&= [\![N]\!]\eta \text{ if } [\![B]\!]\eta \text{ is false} \\
&= \bot \text{ otherwise}
\end{aligned}
$$

This says that the meaning of the entire construct depends only on the meanings of the B, M, and N, and not on their syntax.

Similar meanings are given to other constructs in PCF and other languages, including application and lambda abstraction. This gives the denotational semantics its *compositional* property.

General Questions About Semantics

Is the axiomatic semantics consistent? I.e., is everything equal to everything else? (Also says something about the semantic models.)

Equivalence of semantics:

$$(\forall programs M)(\forall results N) M =_{ax} N \text{ iff } M =_{den} N \text{ iff } M =_{op} N.$$

- where $=_{op}$ means equivalent in every context in op sem
- and $=_{den}$ means they have the same semantic meaning in the denotational semantics
- and $=_{ax}$ means provably equal using the axioms and inference rules in the axiomatic semantics.

In general, operational equivalence is the coarsest ($=_{den} \subseteq =_{op}$), so the interesting question between *op* and *den* is whether every program that can be distinguished in *den* can also be distinguished in *op*. If so, the denotational semantics is call *fully abstract*.

A Small Language Example: Postfix

Modelled after the postscript language, *Postfix* is a simple stack-oriented language. The syntax is given below:

Syntactic Domains

P ∈ Program
Q ∈ Command-Sequence
C ∈ Command
A ∈ Arithmetic-operator = `{add, sub, mul, div}`
R ∈ Relational-operator = `{lt, gt, eq}`
N ∈ Intlit = `{..., -2, -1, 0, 1, 2, ...}`

Production Rules

P ::= (Q)	[Program]
Q ::= C *	[Command-sequence]
C ::= N | (Q) | `pop` | `swap` | A | R	
| `exec`	[Execute]
| `if`	[Conditional]

Postfix Semantics (Informal)

`add, sub, mul, div` `lt, gt, eq`	replace top two values on the stack by the result of applying the operation

e.g., `(1 2 add) => 3`

`swap` swap top two elements

`pop` pop the top element

`(2 3 4 swap sub add) => 3`

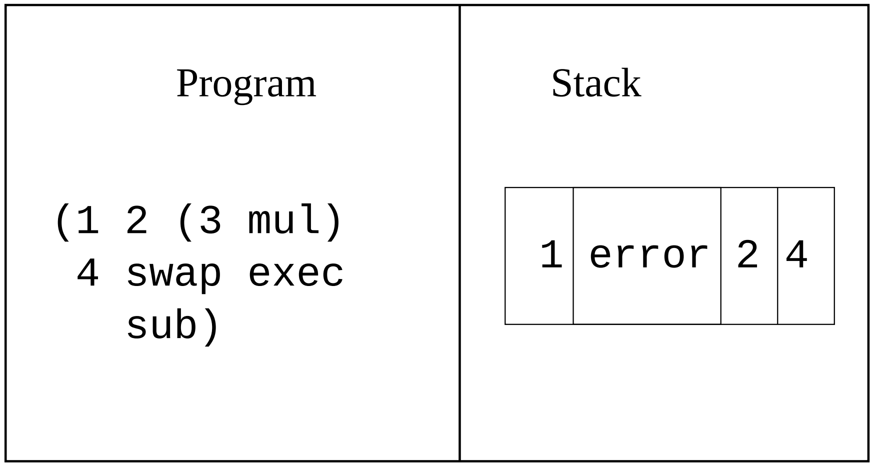
`(C1,...,Cn)` push the executable sequence

`exec` pop and execute the top element

Postfix Operational Semantics

S ϵ Stack = Stackable-Value*
V ϵ Stackable-Value = Intlit + Command-Sequence
A ϵ Answer = Stackable-Value + Error
Error = {error}

Γ = Program x Stack



Γ is the set of "states" or "configurations" on which the abstract machine operates.

Postfix Operational Semantics

< (N . Q), S > => < (Q), N . S > *[numeral]*

< (pop . Q), V . S > => < (Q), S > *[pop]*

< (swap . Q), V1 . V2 . S > => < (Q), V2 . V1 . S > *[swap]*

Evaluation order is: *left-to-right*

Stack grows: *right-to-left (this is internal to the semantics, not significant in the meaning of the language)*

Postfix Operational Semantics

The semantics of conditional in Postfix

< (if . Q_rest), (Q_false) . V_ignore . 0 . S) > =>
< (Q_false @ Q_rest), S >

< (if . Q_rest), V_ignore . (Q_true) . N_test . S) > =>

< (Q_true @ Q_rest), S >

where N_test $=\neq$ 0

Postfix Operational Semantics

$$\frac{\langle (Q\_exec),\ S \rangle \ \text{->}\ \langle (Q\_exec'),\ S' \rangle}{\langle (exec\ .\ Q\_rest),\ (Q\_exec)\ .\ S \rangle \ \text{->}\ ((exec\ .\ Q\_rest),\ (Q\_exec').S' \rangle} \qquad \textit{[exec-progress]}$$

$$\langle (exec\ .\ Q\_rest),\ ()\ .\ S \rangle \quad \text{->} \quad \langle (Q\_rest),\ S \rangle \qquad \textit{[exec-done]}$$

An alternate rule was proposed in class:

$$\langle (exec\ .\ Q\_rest),\ \ (Q)\ .\ S \rangle \quad \text{->} \quad \langle (Q\ @\ Q\_rest)\ ,\ \ S \rangle$$