Principles of Programming Languages

Graham Hutton

Department of Computer Science
University of Nottingham

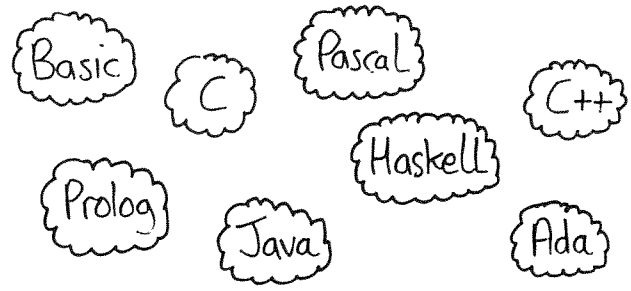
Principles of Programming Languages

Graham Hutton

Lecture 1 - Introduction

BACKGROUND

There are many different programming languages.



Most courses focus on practical aspects of programming in particular languages, such as:

- Language features;
- Algorithms and data structures;
- Space and time efficiency;
- Design methods.

This course takes a more abstract approach, focussing on some of the mathematical concepts underlying programming languages in general:

- Syntax;
- Operational semantics;
- Denotational semantics;
- Lambda calculus;
- Type theory;
- Domain theory.

The central topic of the course is semantics.

WHAT IS SEMANTICS?

Here are a few definitions:

- Oxford Minidictionary:

  "Study of meaning";

- Collins Gem English Dictionary:

  "Study of linguistic meaning";

- Webster's New Dictionary:

  "Science of the meaning of words".

WHERE IS SEMANTICS STUDIED?

Here are the three main areas:

- Linguistics and Philosophy:

  The meaning of sentances in a natural language (e.g. English);

- Mathematics:

  The meaning of terms in a formal logic (e.g. predicate logic.);

✸ Computing Science

  The meaning of programs in a programming language (e.g. C.).

WHY SEMANTICS MATTERS

Program semantics are useful in a number of areas of computing, including:

- Documentation

  A formal semantics provides a complete and precise specification of a language for programmers and compiler writers.

- Formal reasoning

  A formal semantics is a prerequisite for proofs about a programming language and programs written in it.

- Language design

  Semantic issues have a growing influence on the design of programming languages.

- Education

  A knowledge of semantics gives a deeper understanding of programming languages, which will be useful when learning new languages, and when comparing different languages.

- Research

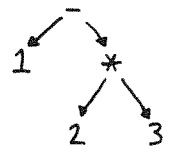
  The elegance of semantics may inspire you to think about applying for a Ph.D. position in programming language research.

COURSE TOPICS

① Syntax

The syntactic structure of programs as trees is specified by a grammar, e.g.:

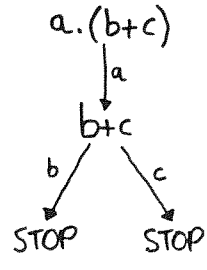
$1 - 2 * 3$ means



② Operational semantics

The meaning of programs as transition systems is specified by transition rules, e.g.:

$a.(b+c)$ means



③ Denotational semantics

The meaning of programs as mathematical functions is specified by recursion equations, e.g.:

$$x := 1 \quad \text{means} \quad f(s) = \text{update}(s, x, 1).$$

④ Lambda calculus

A theory of functions used to study many foundational issues in semantics, e.g.:

$$\begin{aligned} \text{true}\ x\ y &= x \\ \text{false}\ x\ y &= y \\ \text{not}\ x &= x\ \text{false}\ \text{true}. \end{aligned}$$

⑤ Domain theory

Solves the foundational problems with the semantics of recursive types and values, e.g.:

```
data List     = Nil | Cons Int List

len           :: List → Int
len Nil       = 0
len (Cons n ns) = 1 + len ns .
```


COURSE MATERIAL

- Lecture notes: copies of my slides.
- Textbook: none required, but any book on the semantics, principles, or theory of programming languages will be useful for background reading.

ASSESSMENT

- Non-assessed exercises for each lecture;
- One written examination (100%).

HASKELL

Throughout the course, the language Haskell will be used for implementing semantic concept

- Haskell is well-suited to the kind of symbolic manipulation used in semantics;
- Implementing the concepts helps to make them easier to understand, and also allows them to be executed;
- Automatic type checking in Haskell helps to avoid mistakes in our definitions;
- Lecture 2 revises the basics of Haskell.

A TASTE OF SEMANTICS

In most languages, the Booleans (false and true) are built-in to the language.

Question: is this theoretically necessary?

Answer: no, they can be encoded in a simple way using functions!

The type of Booleans:

```
type Bool = a → a → a
```


The basic values:

```
true, false :: Bool
true  x y   = x
false x y   = y
```

The "logical negation" function:

```
not   :: Bool → Bool
not x = x false true
```

Examples:

```
not false = false false true
          = true

not true  = true false true
          = false
```


EXERCISES

① In a similar manner to the not example, define a "logical and" function:

```
and :: Bool → Bool → Bool
```

② Show that your definition has the expected behaviour for all possible combinations of Boolean arguments.

③ Repeat both exercises above for the "logical or" function:

```
or :: Bool → Bool → Bool
```

Principles of Programming Languages

Graham Hutton

Lecture 2 - Review of Haskell

INTRODUCTION

This course assumes some experience with the functional programming language Haskell.

For example, see:

> Lecture notes in functional programming,
> Graham Hutton, Univ. of Nottingham, 1997.

We'll be using the Hugs 1.3 version of Haskell, available for most systems from:

> http://www.cs.nott.ac.uk/
> Department/Staff/mpj/hugs.html.

This lecture reviews the basics of Haskell.

STARTING HUGS FROM UNIX

hugs	Start Hugs;
hugs ⟨filenames⟩	Start and load files.

USEFUL COMMANDS WITHIN HUGS

:l ⟨filenames⟩	Load files;
:r	Repeat last load;
:a ⟨filenames⟩	Load extra files;
:e ⟨filename⟩	Edit file;
:e	Edit last file;
:t ⟨expression⟩	Type an expression;
:?	List commands;
:q	Quit

LAYOUT

In a sequence of definitions, each definition must begin in precisely the same column.

```
a = b+c
      where
        b=10
        c=15-5
d = a*2
```

means

```
{a = b+c
      where
        {b=10;
         c=15-5
 d = a*2}
```

COMMENTS

```
-- A single-line comment.

{- A multiple-line comment,
   which can be nested.  -}
```

CONDITIONALS

```
abs x = if x >=0 then x else -x
```

GUARDS

```
signum x | x<0  = -1
         | x==0 = 0
         | x>0  = 1
```

PATTERN MATCHING

```
not False = True
not True  = False
```

CASE ANALYSIS

```
not x = case x of
          False → True
          True → False
```


LIST COMPREHENSIONS

```
pairs xs     = [(x,y) | x ← xs, y ← xs]
positives xs = [n | n ← ns, n>0]
concat xss   = [x | xs ← xss, x ← xs]
```

ABBREVIATIONS

`f x y z` means `((f x) y) z`

`a → b → c → d` means `a → (b → (c → d))`

`[1,2,3,4]` means `1:(2:(3:(4:[])))`

`"Hello"` means `['H','e','l','l','o']`

BASIC TYPES

Bool	Booleans: False True
Int	Integers: ... -1 0 1 ...
Char	Characters: ... 'a' 'b' ...
a b c ...	Variables

COMPOUND TYPES

[a]	Lists
(a,b)	Pairs
a → b	Functions

TYPE DEFINITIONS

```
type String = [Char]
type Table k v = [(k,v)]
```

DATA DEFINITIONS

```
data Bool   = False | True
data Result = Error | OK Int
data List a = Nil | Cons a (List a)
```

OVERLOADING

```
(+) :: Num a ⇒ a → a → a
(==) :: Eq a ⇒ a → a → Bool
```

RECURSION

```
fac       :: Int → Int
fac 0     = 1
fac (n+1) = (n+1) * fac n

length       :: [a] → Int
length []    = 0
length (x:xs) = 1 + length xs

flatten          :: Tree a → [a]
flatten (Leaf x) = [x]
flatten (Node l r) = flatten l ++ flatten r
```


INPUT/OUTPUT

The library file "IO.hs" defines:

IO a - the type of interactive programs that return results of type a.

together with:

```
getChar   :: IO Char
getLine   :: IO String
putChar   :: Char → IO ()
putStr    :: String → IO ()
putStrLn  :: String → IO ()
sequence  :: [IO a] → IO ()
return    :: a → IO a
```


Examples

```
getChar :: IO Char
getChar = do x ← getCh
             putChar x
             return x

putStr       :: String → IO ()
putStr []    = return ()
putStr (x:xs) = do putChar x
                   putStr xs

getLine :: IO String
getLine = do x ← getChar
             if x == '\n' then
                return ""
             else
                do xs ← getLine
                   return (x:xs)
```


LIBRARY FUNCTIONS

For reference, here are some of the most commonly used library functions:

Booleans

```
(&&) :: Bool → Bool → Bool
(||) :: Bool → Bool → Bool
not  :: Bool → Bool
```

Lists

```
(:)  :: a → [a] → [a]
(++) :: [a] → [a] → [a]
head :: [a] → a
tail :: [a] → [a]
last :: [a] → a
init :: [a] → [a]
```

```
(!!)     :: [a] → Int → a
null     :: [a] → Bool
length   :: [a] → Int
elem     :: Eq a ⇒ a → [a] → Bool
reverse  :: [a] → [a]
zip      :: [a] → [b] → [(a,b)]
```

Pairs

```
fst      :: (a,b) → a
snd      :: (a,b) → b
```

Higher-order functions

```
(.)      :: (b→c) → (a→b) → (a→c)
map      :: (a→b) → [a] → [b]
filter   :: (a→Bool) → [a] → [a]
```


EXAMPLE

Let us write a program for displaying a table showing the distribution of a list of integer exam marks in the range 0-99.

For example:

```
? showtable [57,68,41,61,...
 0:
10: *
20:
30: ****
40: ***********
50: **********************
60: ****************
70: ******
80: ***
90:
```


Displaying a mark:

```
showmark     :: Int → IO()
showmark 0 = putStr " 0"
showmark n = putStr (show n)
```

Displaying a line of a table:

```
showline          :: Int → [Int] → IO()
showline m ms =
  do showmark m
     putStr ": "
     putStrLn stars
  where
     stars = ['*' | _ ← [1..num]]
     num   = length (filter valid ms)
     valid x = (m <= x) && (x <= m+9)
```

Displaying a table:

```
showtable    :: [Int] → IO()
showtable ms = sequence [showline (m*10) ms | m ← [0..9]]
```


EXERCISES

1. Modify the showtable program to also display the min, max, and average mark.

2. Define as many of the library functions from slides 2.11 and 2.12 as you can, without looking up their definitions.

   Hint: when testing your definitions in Hugs, you'll need to rename the functions to avoid clashing with the built-in definitions.

Principles of Programming Languages

Graham Hutton

Lecture 3 - Syntax I

INTRODUCTION

At the lowest level, a program can be viewed simply as a string of characters.

In order to study the meaning of programs at a higher level, we first need to study how programs are formed as strings:

meaning = semantics;

form = syntax.

The next two lectures introduce the basic mathematics underlying program syntax.

LOGIC

We begin by reviewing the operators of propositional (or Boolean) logic:

Notation	Meaning
$A \wedge B$	"and";
$A \vee B$	"or";
$\neg A$	"not";
$A \Rightarrow B$	"implies";
$A \Leftrightarrow B$	"equivalent to".

Note: the $\Leftrightarrow$ operator is sometimes read as "if and only if", abbreviated by "iff".

Predicate logic extends propositional logic by introducing two quantifiers:

Notation	Meaning
$\forall x.\ P(x)$	"for all";
$\exists x.\ P(x)$	"there exists".

Examples

$\forall x.\ (Man(x) \Rightarrow Person(x))$ means

"Every man is a person";

$\exists x.\ (Person(x) \wedge \neg Man(x))$ means

"Not every person is a man".

SETS

A set is a collection of objects, such that:

1. The order of objects is not important;
2. Duplicate objects are not important.

The objects are usually called "elements" or "members".

Notation	Meaning
$\emptyset$	"empty set";
$\{a_1, a_2, \ldots, a_n\}$	"explicitly defined set";
$\{x \mid P(x)\}$	"implicitly defined set";
$x \in S$	"is a member of";
$x \notin S$	"is not a member of".

Useful operations on sets:

Union : $S \cup T = \{x \mid x \in S \vee x \in T\}$

Intersection : $S \cap T = \{x \mid x \in S \wedge x \in T\}$

Subset : $S \subseteq T \Leftrightarrow \forall x.\ x \in S \Rightarrow x \in T$

Equality : $S = T \Leftrightarrow S \subseteq T \wedge T \subseteq S$

Product : $S \times T = \{(x,y) \mid x \in S \wedge y \in T\}$

Powerset : $\mathbb{P}(S) = \{T \mid T \subseteq S\}$

We can also extend $\cup$ and $\cap$ to sets of sets:

$$\bigcup X = \{x \mid \exists S.\ S \in X \wedge x \in S\}$$

$$\bigcap X = \{x \mid \forall S.\ S \in X \Rightarrow x \in S\}$$

STRINGS

A string over a finite set S is a finite sequence of elements of S, with duplicates permitted.

Notation	Meaning
ε	"empty string";
$a_1 a_2 \ldots a_n$	"explicitly defined string";
$s \frown t$	"concatenation".

The underlying set of a string is usually called the "alphabet" of the string.

Examples

1011 and 00101 are both strings over the alphabet $\{0,1\}$ of binary digits.

Useful sets of strings

A^n : All strings of length n over the alphabet A;

A^+ : All non-empty strings over A;

A^* : All strings over A.

Examples

$\{0,1\}^0 = \{\varepsilon\}$

$\{0,1\}^1 = \{0,1\}$

$\{0,1\}^2 = \{00, 01, 10, 11\}$

$\{0,1\}^+ = \{0, 1, 00, 01, 10, 11, \ldots\}$

$\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, \ldots\}$

Formally, A^n can be defined recursively by:

$$A^0 = \{\varepsilon\}$$
$$A^{n+1} = \{x \frown s \mid x \in A \wedge s \in A^n\}.$$

In turn, A^+ and A^* are defined by:

$$A^+ = A^1 \cup A^2 \cup A^3 \cup \ldots = \bigcup_{n=1}^{\infty} A^n$$

$$A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \ldots = \{\varepsilon\} \cup A^+.$$

Note

A^* is usually called the "Kleene closure" of A, after the logician Stephen Kleene (1909-1994.)

GRAMMARS

A grammar is a finite set of rules that describes a (usually infinite) set of strings.

Example: the grammar

number ::= ε | digit number

digit ::= 0 | 1

can be read as

> "A number is either the empty string ε or a digit followed by a number; a digit is either 0 or 1."

and describes the set $\{0,1\}^*$ of binary numbers.

Formally, a (context-free) grammar is a 4-tuple

$$\langle N, T, S, P \rangle$$

where

- N is a finite set of non-terminal symbols;
- T is a finite set of terminal symbols, such that N and T are disjoint, i.e. $N \cap T = \emptyset$;
- $S \in N$ is the start symbol for the grammar;
- $P \subseteq N \times (N \cup T)^*$ is a finite set of production rules for the grammar.

Notes

- The non-terminal symbols N are the names of production rules, and do not appear in strings described by the grammar;
- The terminal symbols T form the alphabet for the strings generated by the grammar;
- The start symbol S is the name of the first production rule to be applied;
- The production rules P are pairs (h,b) where $h \in N$ is a non-terminal symbol called the "head" of the rule, and $b \in (N \cup T)^*$ is a string of symbols called the "body".

BNF NOTATION

Grammars are usually defined using BNF notation:

- BNF stands for Backus-Naur Form, after the two pioneers of the notation;
- Production rules $(h,b) \in P$ are written as:

  $h ::= b$ or $h \to b$;
- A set of production rules

  $h ::= b_1$
  $h ::= b_2$
  $\vdots$
  $h ::= b_n$

  is abbreviated using an "or" operator:

  $h ::= b_1 \mid b_2 \mid \ldots \mid b_n$

EXAMPLE

The binary number grammar:

number ::= ε | digit number

digit ::= 0 | 1

is formalised by the 4-tuple (N,T,S,P) where

$N = \{\text{number}, \text{digit}\}$

$T = \{0,1\}$

$S = \text{number}$

$P = \{(\text{number}, \varepsilon),$
$(\text{number}, \text{digit number}),$
$(\text{digit}, 0),$
$(\text{digit}, 1)\}$

EXERCISES

① Translate the following into English:

a) $\forall x \in \mathbb{N}.\ (\text{Even}(x) \vee \text{Odd}(x))$;

b) $\forall x \in \mathbb{Z}.\ \exists y \in \mathbb{Z}.\ y > x$.

② Translate the following into predicate logic:

a) Some programs have bugs;

b) The smallest natural number is 0.

③ Evaluate the following set-expressions:

a) $\{1,2,3\} \times \{1,2,3\}$;

b) $\{1,2,3\}^2$;

c) $\bigcup(\mathbb{P}\{1,2,3\})$.

④ Write down the first few values of:

a) $\{1,2,3\}^+$;

b) $\{1,2,3\}^*$.

⑤ Using BNF notation, define a grammar that describes lists of single digits in Haskell, such as [3,7,2,1].

Hint: take care that your grammar only describes valid lists.

⑥ Formalise the lists grammar as a 4-tuple.

Principles of Programming Languages

Graham Hutton

Lecture 4 - Syntax II

DERIVATIONS

A string is derivable within a grammar if it can be obtained from the start symbol by applying a sequence of production rules.

Example: given the grammar

number ::= ε | digit number

digit ::= 0 | 1

the string 10 is derivable as follows:

number $\Rightarrow$ digit number
$\Rightarrow$ digit digit number
$\Rightarrow$ digit digit
$\Rightarrow$ 1 digit
$\Rightarrow$ 1 0

Notation: given a grammar $G = \langle N, T, S, P \rangle$,

$$x \underset{G}{\Rightarrow} y$$

means that the string $y \in (N \cup T)^*$ is derivable from the string $x \in (N \cup T)^*$ by applying a single production rule from G to a single occurrence of a non-terminal symbol in x.

Note

The grammar G in $\underset{G}{\Rightarrow}$ is usually omitted when it is clear from the context.

Examples

digit digit $\Rightarrow$ 1 digit ✓

digit digit $\Rightarrow$ digit 0 ✓

digit digit $\Rightarrow$ 1 0 ✗

Formally, given $G = \langle N, T, S, P \rangle$ the relation $\underset{G}{\Rightarrow}$ on strings $(N \cup T)^*$ can be defined by:

$$x \underset{G}{\Rightarrow} y \Leftrightarrow \exists s, t \in (N \cup T)^*. \; \exists (h, b) \in P. \; x = s\,h\,t \wedge y = s\,b\,t.$$

Example: the derivation

digit number $\Rightarrow$ digit digit number

is witnessed by the assignments

s = digit;

t = ε;

h = number;

b = digit number.

Useful relations on strings:

$x \overset{n}{\underset{G}{\Rightarrow}} y$: y is derivable from x by applying precisely n rules;

$x \overset{+}{\underset{G}{\Rightarrow}} y$: y is derivable from x by applying one or more rules;

$x \overset{*}{\underset{G}{\Rightarrow}} y$: y is derivable from x by applying zero or more rules.

Examples (from slide 4.1)

digit digit $\overset{2}{\Rightarrow}$ 10

number $\overset{+}{\Rightarrow}$ 10

number $\overset{*}{\Rightarrow}$ number

Formally, $\overset{n}{\underset{G}{\Rightarrow}}$ can be defined recursively by:

$x \overset{0}{\underset{G}{\Rightarrow}} y \iff x = y$

$x \overset{n+1}{\underset{G}{\Rightarrow}} y \iff \exists z.\ x \underset{G}{\Rightarrow} z \wedge z \overset{n}{\underset{G}{\Rightarrow}} y.$

In turn, $\overset{+}{\underset{G}{\Rightarrow}}$ and $\overset{*}{\underset{G}{\Rightarrow}}$ are defined by:

$x \overset{+}{\underset{G}{\Rightarrow}} y \iff \exists n \in \mathbb{N}.\ x \overset{n+1}{\underset{G}{\Rightarrow}} y;$

$x \overset{*}{\underset{G}{\Rightarrow}} y \iff \exists n \in \mathbb{N}.\ x \overset{n}{\underset{G}{\Rightarrow}} y.$

Notes

- $\overset{+}{\underset{G}{\Rightarrow}}$ is the transitive closure of $\underset{G}{\Rightarrow}$;
- $\overset{*}{\underset{G}{\Rightarrow}}$ is the reflexive/transitive closure of $\underset{G}{\Rightarrow}$.

TERMINOLOGY

Let $G = \langle N, T, S, P \rangle$ be a grammar. Then,

① A string $x \in (N \cup T)^*$ that is derivable from the start symbol, i.e. $S \overset{*}{\underset{G}{\Rightarrow}} x$, is called

"A sentential form of G";

② A string $x \in T^*$ of terminal symbols that is a sentential form of G is called

"A sentance of G";

③ The set $L(G) = \{x \mid x \in T^* \wedge S \overset{*}{\underset{G}{\Rightarrow}} x\}$ of all sentances of G is called

"The language generated by G";

④ If another grammar G' generates the same language as G, then G and G' are called

"Equivalent grammars".

Examples (for the number grammar)

- digit 1 number is a sentential form;
- digit number digit is not a sentential form;
- 101 is a sentance;
- 1 digit 1 is not a sentance;
- The generated language is the set $\{0,1\}^*$;
- An equivalent grammar is:

  number ::= ε | number digit

  digit ::= 0 | 1.

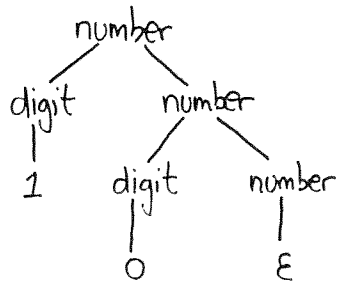
DERIVATION TREES

It is often more useful to represent a derivation of a string as a tree.

Example: the derivation

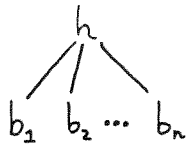
number ⇒ digit number
⇒ digit digit number
⇒ digit digit
⇒ 1 digit
⇒ 1 0

can be represented by the tree



Formally, such trees are constructed as follows:

- The root is the start symbol of the grammar;
- Each branch



corresponds to an application of the production

$h ::= b_1\ b_2 \cdots b_n$

in the derivation, where each b_i is a symbol;

- The leaves are the symbols of the derived string, in left-to-right order.

Note

Derivation trees are also called "parse trees".

While each derivation induces a single tree, in general each such tree represents many different derivations of the same string.

Example: the derivation

number ⇒ digit number
⇒ 1 number
⇒ 1 digit number
⇒ 1 0 number
⇒ 1 0

is also represented by the tree on slide 4.8.

Note

Each derivation tree represents all the derivations of a string that only differ in the order of application of the production rules.

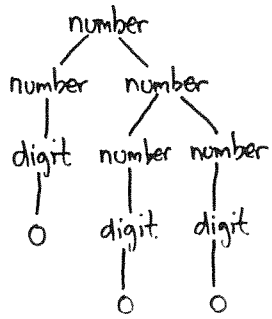
AMBIGUITY

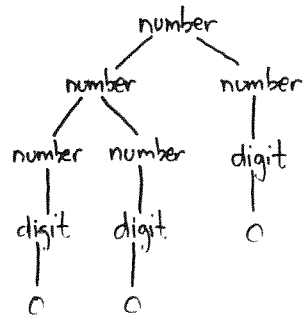
A grammar is unambiguous if every sentance has a single derivation tree, and ambiguous otherwise.

Examples: both grammars for binary numbers given earlier in this lecture are unambiguous, while the following (equivalent) grammar is ambiguous:

number ::= ε | digit | number number
digit ::= 0 | 1.

For example, here are two trees for 000:





EXAMPLE

Let us develop a grammar for simple arithmetic expressions, made up of:

- Single digits 0...9;
- The operators + and *;
- Parentheses (and).

Our first attempt is as follows:

expr ::= expr + expr
| expr * expr
| (expr)
| digit

digit ::= 0 | 1 | ... | 9.

This grammar generates the correct language, but suffers from two ambiguity problems:

① * has higher priority than +, but the grammar does not reflect this.

Example: 1+2*3 has two derivation trees, corresponding to (1+2)*3 and 1+(2*3).

Solution: modify the grammar as follows:

expr ::= expr + expr | term

term ::= term * term | factor

factor ::= (expr) | digit.

Now, 1+2*3 has a single derivation tree.

② + and * are both associative, but this makes the grammar ambiguous.

Example: 1+2+3 has two derivation trees, corresponding to (1+2)+3 and 1+(2+3).

Solution: modify the grammar as follows:

expr ::= expr + term | term

term ::= term * factor | factor

factor ::= (expr) | digit.

Now, the grammar is unambiguous.

Note: to ensure that programs have a single semantics, the grammars for programming languages are usually (for the most part) unambiguous.

EXERCISES

① Write down two further derivations of 10 using the grammar on slide 4.1.

② Define $\overset{*}{\Rightarrow}$ in terms of $\overset{+}{\Rightarrow}$.

③ a) Draw the two derivation trees for 1+2*3 using the grammar on slide 4.12, and the two for 1+2+3 using the grammar on slide 4.13.

b) Draw the single trees for both sentences using the grammar on slide 4.14.

④ Extend the expression grammar to handle exponentiation ^, which has higher priority than + and *, and associates to the right.

Principles of Programming Languages

Graham Hutton

Lecture 5 - Operational Semantics I

INTRODUCTION

One approach to specifying the meaning of programs is operational semantics.

Basic idea:

> How programs execute is specified abstractly in terms of simple transitions.

Example:

$$(1+2)*3 \longrightarrow (3)*3 \longrightarrow 3*3 \longrightarrow 9$$

The next three lectures introduce the basic principles of operational semantics, and show how they can be implemented in Haskell.

EXPRESSIONS

Let us begin by considering the operational semantics of simple arithmetic expressions, given by the following grammar:

expr ::= expr + term | term

term ::= term * factor | factor

factor ::= (expr) | value | variable

where

value ::= ... | -1 | 0 | 1 | ...

variable ::= A | B | C | ...

Intuitively, such expressions can be evaluated to an integer value by using a sequence of transitions, where each transition:

> Replaces a single application of an operator to integer values by the appropriate result (e.g. $1+2 \longrightarrow 3$);

or

> Replaces a single occurrence of a variable by its value (e.g. $A \longrightarrow 3$).

Example: (given $A = 3$)

$$(1+2)*A \longrightarrow (3)*A \longrightarrow 3*A \longrightarrow 3*3 \longrightarrow 9$$

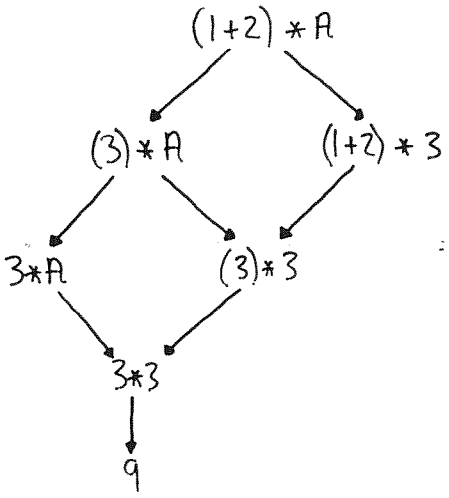
But, other sequences are possible too...

$(1+2)*A \longrightarrow (3)*A \longrightarrow (3)*3 \longrightarrow 3*3 \longrightarrow 9$

or

$(1+2)*A \longrightarrow (1+2)*3 \longrightarrow (3)*3 \longrightarrow 3*3 \longrightarrow 9$

These three transition sequences can be combined into a single transition graph for $(1+2)*A$:



TRANSITION SYSTEMS

Formally, a transition graph (usually called a transition system) is a pair

$$\langle S, \rightarrow \rangle$$

where

- S is a set of "states";
- $\rightarrow \subseteq S \times S$ is a set of "transitions".

Notes

- We abbreviate $(x,y) \in \rightarrow$ by $x \rightarrow y$;
- We read $x \rightarrow y$ as "there is a transition from state x to state y".

Example: (given A=3)

The transition system for the expression $(1+2)*A$ is given by the pair $\langle S, \rightarrow \rangle$, where:

$$S = \{ (1+2)*A,\ (3)*A,\ (1+2)*3,\ 3*A,\ (3)*3,\ 3*3,\ 9 \}$$

$$\rightarrow = \{ ((1+2)*A,\ (3)*A),\ ((1+2)*A,\ (1+2)*3),\ ((3)*A,\ 3*A),\ ((3)*A,\ (3)*3),\ ((1+2)*3,\ (3)*3),\ (3*A,\ 3*3),\ ((3)*3,\ 3*3),\ (3*3,\ 9) \}$$

PARSED EXPRESSIONS

An operational semantics for expressions is a transition system that specifies the possible evaluation steps for all expressions.

It is convenient to define the semantics for parsed expressions, defined in Haskell by:

```
data Expr = Add Expr Expr
          | Mult Expr Expr
          | Paren Expr
          | Val Int
          | Var Char
          deriving Show
```

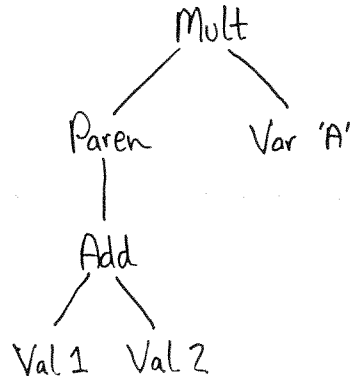
Example: the expression

(1+2) * A

can be parsed to give

Mult (Paren (Add (Val 1) (Val 2))) (Var 'A')

which can be pictured as the tree



Note: parsing is an important topic in computing science, but we don't persue it further here.

STORES

The meaning of variables in an expression will be given by a store that contains the current value of each variable

We represent a store by a list:

```
type Store = [(Char, Int)]
```

and define three operations on stores:

```
empty          :: Store
empty          = []

find           :: Char → Store → Int
find a s       = head [n | (b,n) ← s, a == b]

update         :: Char → Int → Store → Store
update a n s   = (a,n) : [(b,m) | (b,m) ← s, a /= b]
```


Example: given the store

```
s = [('A',1), ('B',2), ('C',3)]
```

we have:

```
? find 'B' s
2

? find 'D' s
Program error: {head []}

? update 'D' 4 s
[('D',4), ('A',1), ('B',2), ('C',3)]

? update 'B' 5 s
[('B',5), ('A',1), ('C',3)]
```


SEMANTICS OF EXPRESSIONS

Now, given a store s, the operational semantics of expressions is formalised by a transition system $\langle S, \rightarrow \rangle$, where:

- S = the set Expr of parsed expressions;
- $\rightarrow$ = the least subset of Expr × Expr satisfying the inference rules below.

In general, we write

$$\frac{A_1 \quad A_2 \quad \dots \quad A_n}{C} \ (R)$$

to mean that R is an inference rule that allows us to conclude that C holds provided that all the assumptions $A_1 \dots A_n$ hold.

The rules for addition:

$$\frac{}{\text{Add (Val n) (Val m)} \longrightarrow \text{Val (n+m)}} \text{ (ADD1)}$$

$$\frac{x \longrightarrow x'}{\text{Add } x\ y \longrightarrow \text{Add } x'\ y} \text{ (ADD2)}$$

$$\frac{y \longrightarrow y'}{\text{Add } x\ y \longrightarrow \text{Add } x\ y'} \text{ (ADD3)}$$

That is, the sum of two values can make the transition to their sum (ADD1), and the sum of two expressions can make a transition if either expression can (ADD2 and ADD3).

The rules for multiplication:

$$\frac{}{\text{Mult (Val n) (Val m)} \longrightarrow \text{Val (n*m)}} \text{ (MULT1)}$$

$$\frac{x \longrightarrow x'}{\text{Mult } x\ y \longrightarrow \text{Mult } x'\ y} \text{ (MULT2)}$$

$$\frac{y \longrightarrow y'}{\text{Mult } x\ y \longrightarrow \text{Mult } x\ y'} \text{ (MULT3)}$$

The rules for parentheses:

$$\frac{}{\text{Paren (Val n)} \longrightarrow \text{Val n}} \text{ (PAREN1)}$$

$$\frac{x \longrightarrow x'}{\text{Paren } x \longrightarrow \text{Paren } x'} \text{ (PAREN2)}$$

The rule for variables:

$$\frac{\text{find } a\ s == n}{\text{Var } a \longrightarrow \text{Val } n} \text{ (VAR)}$$

That is, a variable can make the transition to a value if the variable is currently assigned to the value by the store.

Note

There are no rules for values themselves, because they are already fully evaluated and can make no further transitions.

EXERCISES

1. Draw the transition graph for

   A + (2 * B)

   where A = 1 and B = 3.

2. Extend the Expr type to handle exponentiation, and define transition rules for this operator.

3. Modify the transition rules for addition so that the first expression must be fully evaluated before evaluation of the second can begin.

Principles of Programming Languages

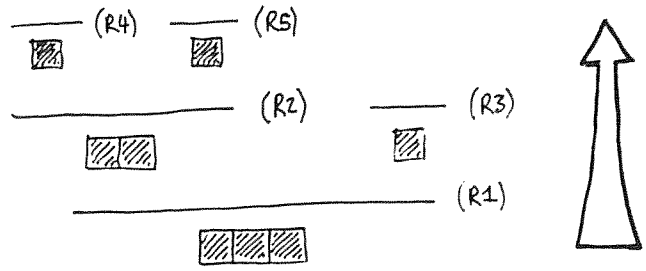
Graham Hutton

Lecture 6 - Operational Semantics II

PROOF TREES

Using inference rules, we can construct proof trees to verify that statements hold.

Here is an (abstract) example:



Such proof trees are built "bottom-up", starting with the statement to be proved, and applying inference rules until there are no assumptions left, or the statements are trivially true.

Example:

Using unparsed notation, the transition

$$(1+2) * A \longrightarrow (3) * A$$

is verified by the following proof tree:

$$\dfrac{\dfrac{\dfrac{}{1+2 \longrightarrow 3}\ (\text{ADD1})}{(1+2) \longrightarrow (3)}\ (\text{PAREN2})}{(1+2) * A \longrightarrow (3) * A}\ (\text{MULT2})$$

That is, we first apply the rule MULT2 to eliminate the use of *, then apply the rule PAREN2 to eliminate the use of ( ), and finally apply the rule ADD1 to verify the use of +.

Example:

Given the state

$$s = [('A', 3)]$$

the alternative transition

$$(1+2) * A \longrightarrow (1+2) * 3$$

is verified by the following proof tree:

$$\dfrac{\dfrac{\text{find 'A' s == 3}}{A \longrightarrow 3}\ (\text{VAR})}{(1+2) * A \longrightarrow (1+2) * 3}\ (\text{MULT3})$$

That is, we first apply the rule MULT3 to eliminate the use of *, and then apply the rule VAR to verify the replacement of the variable by its value.

IMPLEMENTING THE SEMANTICS

Formally deriving the possible transitions for expressions by hand is:

- Laborious - proof trees can be large;
- Error prone - making mistakes is easy.

We now automate the process by implementing the operational semantics of expressions in Haskell.

The implementation also:

- Helps detect errors in the semantics;
- Gives another view of the semantics.

Given a store, the operational semantics of expressions is given by a transition relation

$$\longrightarrow \subseteq \text{Expr} \times \text{Expr}$$

This can be implemented by a Haskell function

```
etrans :: Store → Expr → [Expr]
```

that returns the list of all possible transitions for an expression using a given store.

Example: using unparsed notation

```
etrans [('A',3)] (1+2)*A
```

should give the list

```
[(3)*A , (1+2)*3]
```


The definition of etrans is a simple translation of the inference rules into Haskell notation.

The cases for addition

$$\frac{}{\text{Add (Val } n\text{) (Val } m\text{)} \longrightarrow \text{Val } (n+m)} \text{ (ADD1)}$$

```
etrans s (Add (Val n) (Val m)) = [Val (n+m)]
```

That is, the case for the sum of two values is a direct translation of the rule ADD1.

$$\frac{x \longrightarrow x'}{\text{Add } x\ y \longrightarrow \text{Add } x'\ y} \text{ (ADD2)}$$

+

$$\frac{y \longrightarrow y'}{\text{Add } x\ y \longrightarrow \text{Add } x\ y'} \text{ (ADD3)}$$

```
etrans s (Add x y) =
    [Add x' y | x' ← etrans s x] ++
    [Add x y' | y' ← etrans s y]
```

That is, the case for the sum of two expressions combines the two rules ADD2 and ADD3.

The cases for multiplication

```
etrans s (Mult (Val n) (Val m)) = [Val (n*m)]

etrans s (Mult x y) =
  [Mult x' y | x' ← etrans s x] ++
  [Mult x y' | y' ← etrans s y]
```

The cases for parentheses

```
etrans s (Paren (Val n)) = [Val n]

etrans s (Paren x) =
  [Paren x' | x' ← etrans s x]
```


The case for variables

```
etrans s (Var a) = [Val (find a s)]
```

The case for values

```
etrans s (Val n) = []
```

Example: given

```
s = [('A',3)]
e = Mult (Paren (Add (Val 1) (Val 2))) (Var 'A')
```

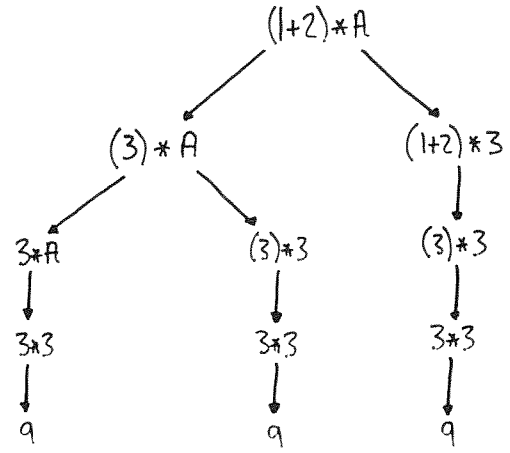
we have:

```
? etrans s e
[Mult (Paren (Val 3)) (Var 'A'),
 Mult (Paren (Add (Val 1) (Val 2))) (Val 3)]
```


SEMANTICS AS TREES

The etrans function generates all the possible transitions for any expression. Often, it is more useful to generate a complete tree that captures all possible transition sequences.

Example: given the state `[('A',3)]` the expression `(1+2)*A` generates the following tree:



We represent trees as follows:

```
data Tree a = Node a [Tree a]
              deriving Show
```

Now, a semantics for expressions as trees can be defined in terms of the transition function

```
etrans :: Store → Expr → [Expr]
```

by simply applying etrans to give a list, and then recursively generating the tree for each element:

```
etree :: Store → Expr → Tree Expr

etree s e =
  Node e [etree s e' | e' ← etrans s e]
```

Example: using unparsed notation,

```
etree [('A',3)] (1+2)*A
```

gives the tree:

```
Node ((1+2)*A)
  [Node ((3)*A)
    [Node (3*A)
      [Node (3*3)
        [Node 9 []]],
    Node ((3)*3)
      [Node (3*3)
        [Node 9 []]]],
  Node ((1+2)*3)
    [Node ((3)*3)
      [Node (3*3)
        [Node 9 []]]]]
```


PROPERTIES OF THE SEMANTICS

It is useful to define the following operation on a transition system:

$x \xrightarrow{*} y$: y can be obtained from x by a sequence of zero or more transitions.

Formally, $\xrightarrow{*}$ can be defined by:

$$x \xrightarrow{*} y \iff \exists n \in \mathbb{N}.\ x \xrightarrow{n} y$$

where

$$x \xrightarrow{0} y \iff x = y$$

$$x \xrightarrow{n+1} y \iff \exists z.\ x \to z \wedge z \xrightarrow{n} y$$

Fact: (for expressions)

In all stores, each expression x evaluates to at most one integer value:

$$\frac{x \xrightarrow{*} \mathrm{Val}\ n \qquad x \xrightarrow{*} \mathrm{Val}\ m}{n == m}$$

Definition:

Two expressions x and y are called equivalent, written $x \sim y$, if they evaluate to the same integer values in all stores:

$$x \sim y \quad \text{iff} \quad x \xrightarrow{*} \mathrm{Val}\ n \iff y \xrightarrow{*} \mathrm{Val}\ n$$

Examples:

$A*B \sim B*A$, but $A*A \not\sim A*2$.

EXERCISES

① Draw the proof trees for

$$A + (2*B) \longrightarrow 1 + (2*B),$$

$$A + (2*B) \longrightarrow A + (2*3)$$

where $A = 1$ and $B = 3$.

The two remaining exercises build upon the corresponding exercises from the previous lecture.

② Extend etrans to handle exponentiation.

③ Modify etrans to implement the new transition rules for addition.

Principles of Programming Languages

Graham Hutton

Lecture 7 - Operational Semantics III

INTRODUCTION

In the last two lectures, we:

- Defined an operational semantics for simple arithmetic expressions;
- Showed how the semantics can be implemented in Haskell.

In this lecture, we:

- Extend the semantics to handle programs in a simple language;
- Extend the implementation accordingly.

PROGRAMS

We consider programs in the language given by the following grammar:

```
prog ::= variable := expr
       | begin prog ; ... ; prog end
       | if expr then prog else prog fi
       | while expr do prog od
       | done
```

Notes

- The logical value False is represented by the integer 0, True by all other integers;
- The trivial program done is for internal use within the semantics only.

Example

The following program calculates the 10th Fibonacci number in the variable A:

```
begin
  A := 1;
  B := 10;
  while B do
    begin
      A := A * B;
      B := B + (-1)
    end.
  od.
end.
```

PARSED PROGRAMS

As for expressions, we define the semantics for parsed programs, defined in Haskell by:

```
data Prog = Assign Char Expr
          | Seqn [Prog]
          | If Expr Prog Prog
          | While Expr Prog
          | Done
          deriving Show
```

Example: (the Fibonacci program)

```
Seqn [Assign 'A' (Val 1),
      Assign 'B' (Val 10),
      While (Var 'B') (Seqn
        [Assign 'A' (Mult (Var 'A') (Var 'B')),
         Assign 'B' (Add (Var 'B') (Val (-1)))])]
```


SEMANTICS OF PROGRAMS

Intuitively, such programs can be executed by a sequence of transitions between pairs of programs and stores, where each transition:

Modifies the program in some way;

and

Possibly modifies the store in some way.

Example: using unparsed notation,

⟨ begin A := 1 ; B := 2 end , [] ⟩

↓

⟨ begin done ; B := 2 end , [('A',1)] ⟩

↓

⟨ begin B := 2 end , [('A',1)] ⟩

↓

⟨ begin done end , [('B',2),('A',1)] ⟩

↓

⟨ begin end , [('B',2),('A',1)] ⟩

↓

⟨ done , [('B',2),('A',1)] ⟩

Formally, the operational semantics of programs is given by a transition system $\langle S, \rightarrow \rangle$, where:

- S = the set Prog × Store of pairs of parsed programs and stores;
- $\rightarrow$ = the least subset of $S \times S$ satisfying the inference rules below.

The rules for assignment:

$$\frac{\text{update } a\ n\ s == s'}{\langle \text{Assign } a\ (\text{Val } n)\ ,\ s\rangle \rightarrow \langle \text{Done}\ ,\ s'\rangle} \quad \text{(ASSIGN1)}$$

That is, the assignment of a value can make the transition to Done, with an updated store.

$$\frac{x\ -s\rightarrow\ x'}{\langle \text{Assign } a\ x\ ,\ s\rangle \rightarrow \langle \text{Assign } a\ x'\ ,\ s\rangle} \quad \text{(ASSIGN2)}$$

That is, the assignment of an expression can make a transition if the expression itself can.

Note: to avoid confusion, we write $-s\rightarrow$ for the transition relation for expressions given a store s.

The rules for sequencing:

$$\frac{}{\langle \text{Seqn } [] , s\rangle \longrightarrow \langle \text{Done} , s\rangle} \text{ (SEQN1)}$$

That is, the sequencing of zero programs can make the transition to Done.

$$\frac{}{\langle \text{Seqn } (\text{Done} : ps) , s\rangle \longrightarrow \langle \text{Seqn } ps , s\rangle} \text{ (SEQN2)}$$

That is, the sequencing of list beginning with Done can make the transition to the tail of the list.

$$\frac{\langle p, s\rangle \longrightarrow \langle p', s'\rangle}{\langle \text{Seqn } (p : ps), s\rangle \longrightarrow \langle \text{Seqn } (p' : ps), s'\rangle} \text{ (SEQN3)}$$

That is, the sequencing of a list can make a transition if the head of the list itself can.

The rules for conditionals:

$$\frac{n \mathrel{/=} 0}{\langle \text{If } (\text{Val } n)\ p\ q , s\rangle \longrightarrow \langle p , s\rangle} \text{ (IF1)}$$

That is, a conditional can make the transition to the first branch if the expression is True.

$$\frac{n == 0}{\langle \text{If } (\text{Val } n)\ p\ q , s\rangle \longrightarrow \langle q , s\rangle} \text{ (IF2)}$$

That is, a conditional can make the transition to the second branch if the expression is False.

$$\frac{x \text{ —} s \rightarrow x'}{\langle \text{If } x\ p\ q , s\rangle \longrightarrow \langle \text{If } x'\ p\ q , s\rangle} \text{ (IF3)}$$

That is, a conditional can make a transition if the argument expression itself can.

The rule for loops:

$$\frac{}{\langle \text{While } x\ p , s\rangle \longrightarrow \langle \text{If } x\ (\text{Seqn } [p, \text{While } x\ p])\ \text{Done} , s\rangle} \text{ (WHILE)}$$

That is, a loop can make the transition to a conditional, such that:

- If the expression evaluates to True, we execute p once and begin the loop again;
- If the expression evaluates to False, we are finished executing the loop.

Note

There are no rules for the trivial program Done, because it is already fully executed, and can make no further transitions itself.

IMPLEMENTING THE SEMANTICS

Even more so than with expressions, deriving the possible transitions for programs by hand is:

- Laborious - proof trees can be huge;
- Error prone - making mistakes is easy.

We now automate the process, by extending the Haskell transition function for expressions,

```
etrans :: Store → Expr → [Expr]
```

by a transition function for programs:

```
ptrans :: (Prog, Store) → [(Prog, Store)]
```

The definition of ptrans is a simple translation of the inference rules into Haskell notation.

The cases for assignments:

$$\frac{\text{update } a\ n\ s == s'}{\langle \text{Assign } a\ (\text{Val } n), s\rangle \rightarrow \langle \text{Done}, s'\rangle}\ (\text{ASSIGN1})$$

```
ptrans (Assign a (Val n), s) =
  [(Done, update a n s)]
```

$$\frac{x - s \rightarrow x'}{\langle \text{Assign } a\ x, s\rangle \rightarrow \langle \text{Assign } a\ x', s\rangle}\ (\text{ASSIGN2})$$

```
ptrans (Assign a x, s) =
  [(Assign a x', s) | x' <- etrans s x]
```


The cases for sequencing:

```
ptrans (Seqn [], s)        = [(Done, s)]
ptrans (Seqn (Done : ps), s) = [(Seqn ps, s)]
ptrans (Seqn (p : ps), s) =
  [(Seqn (p' : ps), s') | (p', s') <- ptrans (p, s)]
```

The cases for conditionals:

```
ptrans (If (Val 0) p q, s) = [(q, s)]
ptrans (If (Val n) p q, s) = [(p, s)]
ptrans (If x p q, s)       =
  [(If x' p q, s) | x' <- etrans s x]
```


The case for loops:

```
ptrans (While x p, s) =
  [(If x (Seqn [p, While x p]) Done, s)]
```

The case for done:

```
ptrans (Done, s) = []
```

SEMANTICS AS TREES

Now, a semantics for programs as trees can be defined by recursively applying the transition function for programs, as follows:

```
ptree :: (Prog, Store) -> Tree (Prog, Store)
ptree ps = Node ps [ptree ps' | ps' <- ptrans ps]
```


EXERCISES

① Try out a few small programs using the functions ptrans and ptree.

② Using ptree, write a program to calculate the number of transitions in the tree for the Fibonacci program on slide 7.4.

Note: the answer is probably a lot bigger than you might think!

Principles of Programming Languages

Graham Hutton

Lecture 8 - Denotational Semantics I

INTRODUCTION

Another approach to specifying the meaning of programs is denotational semantics.

Basic idea:

> What programs mean is specified abstractly in terms of mathematical functions.

Example:

$$[\![(1+2) * 3]\!] = 9$$

The next three lectures introduce the basic principles of denotational semantics, and show how they can be implemented in Haskell.

EXPRESSIONS

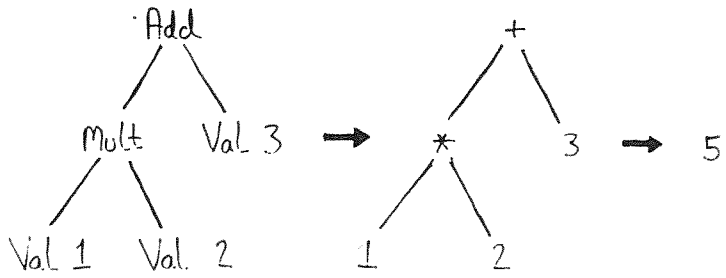
Let us begin by considering the denotational semantics of our simple arithmetic expressions, given by the following datatype:

```
data Expr = Add Expr Expr
          | Mult Expr Expr
          | Paren Expr
          | Val Int
          | Var Char
          deriving Show
```

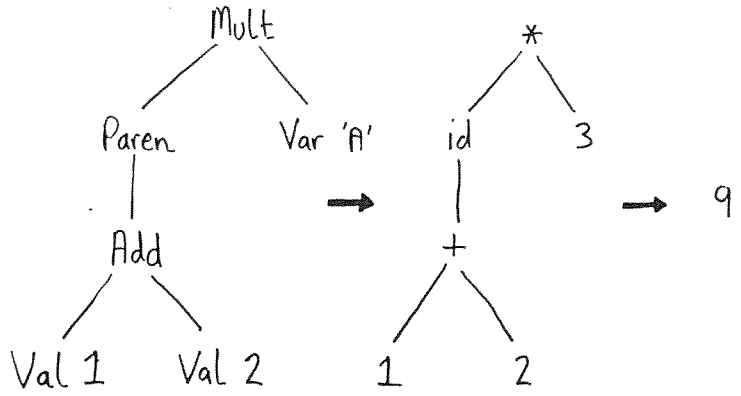
Intuitively, such expressions can be evaluated directly to an integer value by ...

> Replacing each application of Add, Mult, and Paren by the appropriate function on integers;
>
> and
>
> Replacing each value by the underlying integer;
>
> and
>
> Replacing each variable by its integer value.

Example:



Example: (given A = 3)



That is,

Add	is replaced by	+
Mult		*
Paren		id
Val n		n
Var a		find a s

SEMANTICS OF EXPRESSIONS

The denotational semantics of expressions is given by a valuation function

$$[\![\ ]\!] : \text{Expr} \to \text{Store} \to \text{Int}$$

such that $[\![e]\!]s$ is the integer value denoted by the expression e given the store s:

$$[\![\text{Add } x\ y]\!]\ s = [\![x]\!]s + [\![y]\!]s$$

$$[\![\text{Mult } x\ y]\!]\ s = [\![x]\!]s * [\![y]\!]s$$

$$[\![\text{Paren } x]\!]\ s = [\![x]\!]s$$

$$[\![\text{Val } n]\!]\ s = n$$

$$[\![\text{Var } a]\!]\ s = \text{find } a\ s$$

Example: given the store

$$s = [('A', 3)]$$

the denotation (using unparsed notation)

$$[\![\ (1+2) * A\ ]\!]\ s = 9$$

is verified by the following calculation:

$$\begin{aligned}
[\![\ (1+2) * A\ ]\!]\ s &= [\![(1+2)]\!]s * [\![A]\!]s \\
&= [\![1+2]\!]s * [\![A]\!]s \\
&= ([\![1]\!]s + [\![2]\!]s) * [\![A]\!]s \\
&= (1 + [\![2]\!]s) * [\![A]\!]s \\
&= (1 + 2) * [\![A]\!]s \\
&= (1+2) * 3 \\
&= 9
\end{aligned}$$

COMPOSITIONALITY

The valuation function $[\![\ ]\!]$ for expressions is compositional, in the sense that:

> The denotation of each expression is defined purely in terms of the denotations of its subexpressions.

Example:

$$[\![\ \underbrace{\text{Add } x\ y}_{\text{expression}}\ ]\!]\ s = [\![\underbrace{x}_{\text{subexpressions}}]\!]s + [\![\underbrace{y}_{\text{subexpressions}}]\!]s$$

Compositionality of the semantics simplifies reasoning about expressions, because it allows us to substitute "equals for equals."

Example: we can substitute equals for equals within an addition expression:

$$\frac{[\![x]\!] = [\![x']\!] \qquad [\![y]\!] = [\![y']\!]}{[\![\text{Add } x\ y]\!] = [\![\text{Add } x'\ y']\!]}$$

Proof: for any store s,

$$\begin{aligned} & [\![\text{Add } x\ y]\!]_s \\ = \; & [\![x]\!]s + [\![y]\!]s \\ = \; & [\![x']\!]s + [\![y']\!]s \\ = \; & [\![\text{Add } x'\ y']\!]s \end{aligned}$$

DENOTATIONAL SEMANTICS

Formally, a denotational semantics of a language L is given by a pair

$$\langle D, [\![\ ]\!] \rangle$$

where

- D is a set of "semantic values";
- $[\![\ ]\!] : L \to D$ is a "valuation function";

such that:

> The denotation of each L-sentance is defined purely in terms of the denotations of its L-subsentances.

Example:

The denotational semantics for expressions Expr is given by the pair $\langle D, [\![\ ]\!] \rangle$ where:

$$D = \text{Store} \to \text{Int}$$

$$[\![\ ]\!] : \text{Expr} \to (\text{Store} \to \text{Int}) =$$

the function defined on slide 8.5.

Notes

- The semantic values are functions - this is often the case in denotational semantics;
- For more complicated languages, D may be recursively defined, which causes problems. We will return to this later on.

IMPLEMENTING THE SEMANTICS

Because semantic values usually involve functions, and valuation functions are recursive, it is natural to implement denotational semantics in Haskell.

Example: the valuation function

$$[\![\ ]\!] : \text{Expr} \to \text{Store} \to \text{Int}$$

can be implemented by a Haskell function

```
eval :: Expr -> Store -> Int
```

The definition of eval is a direct translation of the definition of $[\![\ ]\!]$ into Haskell notation.

```
eval (Add x y) s = eval x s + eval y s
eval (Mult x y) s = eval x s * eval y s
eval (Paren x) s = eval x s
eval (Val n) s = n
eval (Var a) s = find a s
```

Example: given

```
e = Mult (Paren (Add (Val 1) (Val 2))) (Var 'A')
s = [('A', 3)]
```

we have:

```
? eval e s
9
```


COMPARING THE SEMANTICS

We now have two semantics for expressions:

- An operational semantics: a transition relation $\rightarrow$ defined by inference rules;
- A denotational semantics: a valuation function $[\![\ ]\!]$ defined by recursion.

This raises two natural questions.

① How are the semantics similar?

Two expressions are operationally equivalent precisely when they are denotationally equal:

$$x \sim y \iff [\![x]\!] = [\![y]\!].$$

② How are the semantics different?

The operational semantics is concerned with how an expression evaluates, and is suitable for studying operational issues, such as:

- Reduction order;
- Time complexity;
- Space usage.

The denotational semantics is only concerned with what an expression evaluates to, and is suitable for studying denotational issues, such as:

- Expression equivalence;
- Expression transformation.

EXERCISES

① In Haskell, define a denotational semantics

```
unparse :: Expr → String
```

that converts an expression into a string.

② Recall the function foldr that processes lists by replacing each (:) by a given function, and [] by a given value.

a) Define analogous function foldE that processes expressions by replacing each constructor by a given function.

b) Using foldE, redefine the function

```
eval :: Expr → (Store → Int)
```

Principles of Programming Languages

Graham Hutton

Lecture 9 - Denotational Semantics II

INTRODUCTION

In the last lecture, we:

- Defined a denotational semantics for simple arithmetic expressions;
- Showed how the semantics can be implemented in Haskell.

In this lecture, we:

- Extend the semantics to handle programs in our simple language;
- Extend the implementation accordingly.

PROGRAMS

Recall our datatype for programs:

```
data Prog = Assign Char Expr
          | Seqn [Prog]
          | If Expr Prog Prog
          | While Expr Prog
          | Done
          deriving Show
```

As we saw earlier, such programs can be executed by a sequence of transitions.

Intuitively, such programs can also be executed directly to a final store.

Example: using unparsed notation,

```
begin A:=1 ; B:=2 end
```

↓

```
[('B',2),('A',1)]
```

Example: the Fibonacci program,

```
begin A:=1 ; B:=10 ; while B do
  begin A:=A*B ; B:=B+(-1) end od end
```

↓

```
[('B',0),('A',3628800)]
```

SEMANTICS OF PROGRAMS

Formally, the denotational semantics for programs Prog is given by the pair $\langle D, [\![\ ]\!] \rangle$, where:

D = the set Store → Store of functions from stores to stores;

$[\![\ ]\!]$: Prog → (Store → Store) = the function defined by the cases below.

Note

To avoid confusion, we write $E[\![\ ]\!]$ for the evaluation function for expressions, and $P[\![\ ]\!]$ for the evaluation function for programs.

The case for assignments:

$$P[\![\text{Assign } a\ x]\!]\ s = \text{update } a\ (E[\![x]\!]\ s)\ s$$

That is, an assignment can be executed by first evaluating the expression, and then updating the store with the result.

The cases for sequencing:

$$P[\![\text{Seqn } []]\!]\ s = s$$

That is, the sequence of zero programs can be executed by doing nothing.

$$P[\![\text{Seqn } (p:ps)]\!]\ s = P[\![\text{Seqn } ps]\!]\ (P[\![p]\!]\ s)$$

That is, a non-empty sequence can be executed by first executing the first program, and then recursively executing the remaining programs using the resulting store.

The case for conditionals:

$$P[\![\text{If } x\ p\ q]\!]\ s = \begin{cases} P[\![p]\!]\ s, & E[\![x]\!]\ s \neq 0 \\ P[\![q]\!]\ s, & E[\![x]\!]\ s == 0 \end{cases}$$

That is, a conditional can be executed by first evaluating the expression, and then recursively executing either the first or second branch, depending on the value of the result.

The case for loops:

$$P[\![\text{While } x\ p]\!]\ s =$$
$$P[\![\text{If } x\ (\text{Seqn } [p, \text{While } x\ p])\ \text{Done}]\!]\ s$$

That is, a loop can be executed by executing the appropriate conditional.

Note: this is the same technique used in the operational semantics of loops.

The case for done:

$$P[\![\text{Done}]\!]\ s = s$$

COMPOSITIONALITY

Problem: the valuation function $P[\![\ ]\!]$ for programs is not compositional, because:

> The denotation of a loop is not defined purely in terms of the denotations of its subcomponents.

At present, we have:

$$P[\![\text{While } x\ p]\!] = P[\![\ldots x \ldots p \ldots]\!]$$

To be compositional, we require:

$$P[\![\text{While } x\ p]\!] = \ldots E[\![x]\!] \ldots P[\![p]\!] \ldots$$

To fix the problem, we first define:

$$\text{loop } s = P[\![\text{While } x\ p]\!]\, s$$

Then, it can be shown that:

$$\text{loop } s = \begin{cases} s & , E[\![x]\!]s == 0 \\ \text{loop } (P[\![p]\!]s) & , E[\![x]\!]s \neq 0 \end{cases}$$

That is, the loop function first evaluates the expression x, and then:

- If the result is False, does nothing;
- If the result is True, executes the program p and then recursively applies loop to the resulting store.

Turning things around, we now have:

$$P[\![\text{While } x\ p]\!]\, s = \text{loop } s \text{ , where}$$

$$\text{loop } s = \begin{cases} s & , E[\![x]\!]s == 0 \\ \text{loop } (P[\![p]\!]s) & , E[\![x]\!]s \neq 0 \end{cases}$$

which is a compositional definition, because:

$$P[\![\text{While } x\ p]\!] = \ldots E[\![x]\!] \ldots P[\![p]\!] \ldots$$

Note

The problem is solved by moving the recursion and the conditional from the syntactic level to the semantic level.

IMPLEMENTING THE SEMANTICS

The valuation function

$$P[\![\ ]\!] : \text{Prog} \to \text{Store} \to \text{Store}$$

can be implemented by a Haskell function

```
pval :: Prog -> Store -> Store
```

The definition of pval is a direct translation of the definition of $P[\![\ ]\!]$ into Haskell notation.

```
pval (Assign a x) s = update a (eval x s) s

pval (Seqn [])    s = s
pval (Seqn (p:ps)) s = pval (Seqn ps) (pval p s)
```

```
pval (If x p q) s = if eval x s /=0 then
                      pval p s
                    else
                      pval q s

pval (While x p) s = loop s
  where
    loop s = if eval x s == 0 then
               s
             else
               loop (pval p s)

pval (Done) s      = s
```

Note

A denotational semantics is essentially an interpreter written in a functional language.

EXERCISES

In some programming languages (for example C), the distinction between programs and expressions is removed, such that:

- Programs can return values;
- Expressions can modify the store.

Example:

```
begin
  A := B := 10 ;
  while B := B - 1 do
    A := A + A
  od
end
```


Our existing grammars for programs and expressions can be combined to give a grammar for such a language, as follows:

```
prog   ::= variable := [prog]
         | begin prog ; ... ; prog end
         | if [prog] then prog else prog fi
         | while [prog] do prog od
         | [expr]

expr   ::= expr + term | term

term   ::= term * factor | factor

factor ::= ([prog]) | value | variable
```

The changes are marked with a box [ ].

① Define a datatype Prog for parsed programs in this grammar, by combining the existing datatypes Prog and Expr.

② Define a denotational semantics

$$\langle D, [\![\ ]\!] \rangle$$

for such programs, where:

$$D = Store \to (Int, Store)$$

$$[\![\ ]\!] : Prog \to (Store \to (Int, Store))$$

③ Implement your semantics in Haskell.

Principles of Programming Languages

Graham Hutton

Lecture 10 - Denotational Semantics III

INTRODUCTION

In this lecture, we

- Work through the exercises from lecture 9 on the whiteboard.

Principles of Programming Languages

Graham Hutton

Lecture 11 - Lambda Calculus I

INTRODUCTION

In denotational semantics, programs are given meaning in terms of mathematical functions.

Question:

What is an appropriate basic language for studying such functions themselves?

Answer:

The lambda calculus, developed by Alonzo Church and Haskell Curry in the 1920s - 1940s.

The next three lectures introduce the basic principles of the lambda calculus.

SYNTAX

Expressions in the lambda calculus are built from three basic components:

- Variables

  An arbitrary expression can be represented by a (non-mutable) variable;

- Abstractions

  A function can be defined by abstracting over a variable in an expression;

- Applications

  A function can be applied to an argument expression to yield a result expression.

Formally, the syntax of lambda expressions is given by the following grammar:

```
expr ::= var                  (variables)
       | (λ var . expr)       (abstractions)
       | (expr expr)          (applications)
```

where var is an (infinite) set of variables.

Notes

- The grammar is unambiguous, through the use of parentheses in expressions;
- λ is the Greek letter "lambda";

- Abstractions are nameless functions;
- The lambda calculus is a higher-order language: functions can be freely passed as arguments and returned as results;
- Functions have one argument: functions with more than one argument can be defined as curried functions. For example:

$$(\lambda a . (\lambda b . x)).$$

- In Haskell, $(\lambda a . x)$ is written $\backslash a \rightarrow x$.

ABSTRACTIONS

Intuitively, an abstraction

$$(\lambda a . x)$$

represents the function that when applied to an argument expression y,

$$((\lambda a . x)\ y)$$

yields the expression x with each occurrence of a replaced by y as its result.

(But ... there are subtle complications.)

Examples:

- $(\lambda a . a)$ is the "identity" function that takes an expression a as its argument, and yields a unchanged as its result;
- $(\lambda f . (f\ f))$ is the "self application" function that takes a function f as its argument, and yields f applied to itself as its result;
- $(\lambda a . (\lambda f . (f\ a)))$ is the "reverse application" function that takes an expression a and a function f as its arguments, and yields f applied to a as its result.

PARENTHESES

To reduce the number of parentheses required in lambda expressions, a number of syntactic conventions are normally used:

- The outermost parentheses can be omitted. For example:

$$\lambda a . a \quad \text{means} \quad (\lambda a . a).$$

- Application is assumed to associate to the left. For example:

$$f\ x\ y\ z \quad \text{means} \quad (((f\ x)\ y)\ z).$$

- A sequence of abstractions can be written using a single λ. For example:

  $\lambda\, a\ b\ c\ .\ x$ means $(\lambda a.(\lambda b.(\lambda c.x)))$.

- Application is assumed to have higher precedence than abstraction. For example:

  $\lambda a.\ a\ a$ means $(\lambda a.(a\ a))$
  not $((\lambda a.a)\ a)$;

  $\lambda a.\lambda b.\ a\ b$ means $(\lambda a.(\lambda b.(a\ b)))$
  not $(\lambda a.((\lambda b.a)\ b))$
  or $((\lambda a.(\lambda b.a))\ b)$.

OPERATIONAL SEMANTICS

Intuitively, lambda expressions can be evaluated by using a sequence of transitions, where each transition:

> Replaces a single application of an abstraction to an expression by the appropriate result.

Example:

$(\lambda a.\ \lambda f.\ f\ a)\ x\ g$

$\downarrow$

$(\lambda f.\ f\ x)\ g$

$\downarrow$

$g\ x$

Formally, the operational semantics is given by a transition system $\langle S, \rightarrow \rangle$, where:

S = the set of lambda expressions generated by the expr grammar;

$\rightarrow$ = the least subset of $S \times S$ satisfying the inference rules below.

The rules for abstraction

$$\frac{}{((\lambda a.x)\ y) \longrightarrow x[y/a]}\ (ABS1)$$

That is, the application of an abstraction $(\lambda a.x)$ to an expression y can make the transition to x with each occurrence of a replaced by y, written $x[y/a]$.

$$\frac{x \longrightarrow x'}{(\lambda a.x) \longrightarrow (\lambda a.x')}\ (ABS2)$$

That is, an abstraction can make a transition if its component expression can.

The rules for application

$$\frac{x \longrightarrow x'}{(x\ y) \longrightarrow (x'\ y)}\ (APP1)$$

$$\frac{y \longrightarrow y'}{(x\ y) \longrightarrow (x\ y')}\ (APP2)$$

That is, an application can make a transition if either expression can.

Notes

- There are no rules for variables themselves, because they represent arbitrary expressions, which can make no transitions;
- For (unclear) historical reasons, the ABS1 rule is usually called β-reduction, where β is the Greek letter "beta";
- β-reduction is the key rule - all the other rules simply express that the β-reduction rule can be applied in any context.

Example:

Given the abbreviations:

TRUE = λa. λb. a

FALSE = λa. λb. b

NOT = λf. f FALSE TRUE

we have:

NOT FALSE

= {Definition of NOT}

(λf. f FALSE TRUE) FALSE

→ {β-reduction}

FALSE FALSE TRUE

= {Definition of FALSE}

(λa. λb. b) FALSE TRUE

→ {β-reduction}

(λb. b) TRUE

→ {β-reduction}

TRUE

That is, we have:

NOT FALSE $\xrightarrow{*}$ TRUE

Dually, we also have:

NOT TRUE $\xrightarrow{*}$ FALSE

EXERCISES

① Fully parenthesize the following expressions:

(λa. λb. b) FALSE TRUE

(λb. b) TRUE

② Draw the proof tree for:

(λa. λb. b) FALSE TRUE ⟶ (λb. b) TRUE

③ Without using a recursive abbreviation, find a lambda expression whose evaluation never terminates.

Hint: look for an expression of the form (x x).

Principles of Programming Languages

Graham Hutton

Lecture 12 - Lambda Calculus II

INTRODUCTION

In the last lecture, we:

- Defined the syntax of the lambda calculus;
- Defined an operational semantics for the language in terms of β-reduction.

In this lecture, we:

- Formalise the notion of substitution used in β-reduction;
- Show how it can be implemented in Haskell.

SUBSTITUTION

Informally, we write

$$x\ [y/a]$$

for the result of substituting the expression y for each occurrence of the variable a in the expression x. For example:

$$\begin{aligned} a\ [y/a] &= y \\ b\ [y/a] &= b \\ (a\ a)\ [y/a] &= (y\ y) \\ (a\ b)\ [y/a] &= (y\ b) \end{aligned}$$

But ... there are subtle problems.

BOUND VARIABLES

A particular occurrence of a variable in a lambda expression is called:

- Bound, if it occurs within an abstraction for that variable;
- Free, otherwise.

Example:

$$\lambda a\ .\ (a\ b)$$

bound free

Notes

- A variable can be bound in more than one way in a single expression.

  Example:

  $((\lambda a.a)\ (\lambda a.a))$

  bound bound

- A variable can occur both bound and free in a single expression.

  Example:

  $((\lambda a.a)\ a)$

  bound free

ALPHA EQUIVALENCE

Intuitively, lambda expressions that only differ in their naming of bound variables should be regarded as equal.

Examples:

$$\lambda a.a = \lambda b.b$$
$$\lambda a.\lambda b.ab = \lambda b.\lambda a.ba$$
$$\lambda a.\lambda b.ab \neq \lambda b.\lambda a.ab$$

This kind of equality is called α-equivalence, where α is the Greek letter "alpha".

PROBLEMS

The present notion of substitution does not preserve α-equivalence, due to two problems with the treatment of bound variables.

① Bound variables should not be substituted.

Consider the α-equivalent expressions

$(\lambda a.a)\ a$ and $(\lambda b.b)\ a$.

Applying the substitution $[y/a]$ to each gives

$(\lambda a.y)\ y$ and $(\lambda b.b)\ y$,

which are not α-equivalent.

Solution: modify the definition of substitution to only substitute for free variables:

> We write $x\,[y/a]$ for the result of substituting the expression y for each free occurrence of the variable a in the expression x.

Example: applying $[y/a]$ to

$(\lambda a.a)\ a$ and $(\lambda b.b)\ a$

bound free free

now gives the α-equivalent expressions

$(\lambda a.a)\ y$ and $(\lambda b.b)\ y$.

(2) Bound variables may require renaming.

Consider the α-equivalent expressions

$(\lambda b.\, a)$ and $(\lambda c.\, a)$

(a: free; a: free)

Applying the substitution $[b/a]$ to each gives

$(\lambda b.\, b)$ and $(\lambda c.\, b)$

(b: bound; b: free)

which are not α-equivalent.

Solution: modify the definition of substitution to rename bound variables if required:

In $x\,[y/a]$, bound variables in x should be renamed as required to ensure that no free variable in y becomes bound in the result of $x\,[y/a]$.

Example: applying $[b/a]$ to

$(\lambda b.\, a)$ and $(\lambda c.\, a)$

now gives the α-equivalent expressions

$(\lambda d.\, b)$ and $(\lambda c.\, b)$

(b: renamed; b: unchanged)

FORMALISING SUBSTITUTION

The case for variables:

$$b\,[y/a] = \begin{cases} y, & a = b \\ b, & a \neq b \end{cases}$$

That is, substituting an expression y for the variable a in the variable b gives y if the variables are equal, and b otherwise.

The case for application:

$$(x_1\ x_2)\,[y/a] = (x_1\,[y/a]\ \ x_2\,[y/a])$$

That is, substituting in an application reduces to substituting in the two subexpressions.

The case for abstraction:

$$(\lambda b.\, x)\,[y/a] =$$

$$\begin{cases} \lambda b.\, x & , \quad a = b \\ \lambda b.\, (x\,[y/a]) & , \quad a \neq b \wedge b \notin fv(y) \\ (\lambda b'.\, x')\,[y/a] & , \quad a \neq b \wedge b \in fv(y) \wedge b' \notin fv(x) \wedge b' \notin fv(y) \end{cases}$$

where

- $fv(z)$ = the set of free variables in z;
- $x' = x\,[b'/b]$.

That is, $(\lambda b.x)\ [y/a]$ gives

- $\lambda b.x$ unchanged if $a = b$, since in this case there are no free occurrences of a in $(\lambda b.x)$ to substitute for;
- $\lambda b.(x\ [y/a])$ if $a \neq b$ and there are no free occurrences of b in y, since in this case b does not require renaming;
- $(\lambda b'.x')\ [y/a]$ if $a \neq b$ and there are free occurrences of b in y, since in this case b requires renaming to a new variable b' before substituting.

IMPLEMENTING SUBSTITUTION

Representing expressions, with integers as variables:

```
data Expr = Var Int
          | Abs Int Expr
          | App Expr Expr
          deriving Show
```

Calculating the free variables in an expression:

```
free             :: Expr → [Int]
free (Var a)     = [a]
free (Abs a x)   = remove a (free x)
free (App x y)   = free x ++ free y

remove           :: Eq a ⇒ a → [a] → [a]
remove x xs      = [y | y ← xs, y /= x]
```


Creating a fresh variable:

```
fresh    :: [Int] → Int
fresh as = maximum as + 1
```

The substitution function:

```
subst :: Expr → Expr → Int → Expr

subst (Var b) y a
  | a == b    = y
  | otherwise = Var b

subst (App x1 x2) y a
  = App (subst x1 y a) (subst x2 y a)

subst (Abs b x) y a
  | a == b                  = Abs b x
  | not (elem b (free y))   = Abs b (subst x y a)
  | otherwise               = subst (Abs b' x') y a
    where
      b' = fresh (free x ++ free y)
      x' = subst x (Var b') b
```


EXERCISES

① Identify the free/bound variables in:

$$(\lambda a.\ a\ b)\ \ (\lambda b.\ a\ b).$$

② Using the definition of substitution on slides 12.10 and 12.11, evaluate:

$$((\lambda a.\ a)\ a)\ [b/a];$$

$$(\lambda b.\ b\ a)\ [b/a].$$

③ Check your answers to ② using the Haskell implementation of substitution.

Principles of Programming Languages

Graham Hutton

Lecture 13 - Lambda Calculus III

INTRODUCTION

The λ-calculus is a very primitive language, providing only three programming features:

- Variables (placeholders for expressions);
- Abstractions (nameless functions);
- Application (the "computation" mechanism).

However, despite its simplicity:

> The λ-calculus has the expressive power to represent any datatype, program, or language feature from any (sequential) language.

This lecture shows how a few common datatypes, programs, and language features can be represented.

BOOLEANS

We have already seen how the Booleans can be represented in the λ-calculus:

$$\text{TRUE} = \lambda a.\, \lambda b.\, a$$
$$\text{FALSE} = \lambda a.\, \lambda b.\, b$$

These definitions may seem strange, but they support the definition of functions on Booleans, such as the "logical negation" function:

$$\text{NOT} = \lambda f.\, f\ \text{FALSE}\ \text{TRUE}$$

For example, we have:

$$\text{NOT TRUE} \xrightarrow{*} \text{FALSE}$$
$$\text{NOT FALSE} \xrightarrow{*} \text{TRUE}$$

Notes:

- The representations for TRUE and FALSE can be motivated, but this is beyond the scope of this course;
- The flexibility of the λ-calculus means that there are also a number of "meaningless" interactions, such as:

$$\text{TRUE FALSE} \xrightarrow{*} \lambda b.\, \text{FALSE}$$
$$\text{TRUE NOT} \xrightarrow{*} \lambda b.\, \text{NOT}$$
$$\text{NOT NOT} \xrightarrow{*} \lambda b.\, \text{TRUE}.$$

PAIRS

In the λ-calculus, a pair of expressions (x, y) can be represented as follows:

$$(x,y) = \lambda f.\ f\ x\ y$$

Again, this definition may seem strange, but it supports the definition of functions on pairs, such as the "selection" functions:

$$FST = \lambda p.\ p\ TRUE$$
$$SND = \lambda p.\ p\ FALSE$$

For example, we have: ...

FST (x,y)

= { Definition of FST }

$(\lambda p.\ p\ TRUE)\ (x,y)$

→ { β-reduction }

$(x,y)\ TRUE$

= { Definition of (x,y) }

$(\lambda f.\ f\ x\ y)\ TRUE$

→ { β-reduction }

$TRUE\ x\ y$

→ { Definition of TRUE, β-reduction }

x.

Dually, we also have:

$SND\ (x,y) \xrightarrow{*} y$

NATURAL NUMBERS

There are many ways to represent the natural numbers in the λ-calculus.

The original representation is due to Alonzo Church, and is called the Church numerals:

$$0 = \lambda f.\lambda a.\ a$$
$$1 = \lambda f.\lambda a.\ f\ a$$
$$2 = \lambda f.\lambda a.\ f\ (f\ a)$$
$$3 = \lambda f.\lambda a.\ f\ (f\ (f\ a))$$
$$\vdots$$
$$n = \lambda f.\lambda a.\ \underbrace{f\ (f\ (f \cdots a) \cdots)}_{n \text{ times}}$$

For example, using these definitions the "successor" function on naturals can be defined by:

$$SUCC = \lambda n.\lambda f.\lambda a.\ n\ f\ (f\ a)$$

Example:

SUCC 0

= { Definition of SUCC }

$(\lambda n.\lambda f.\lambda a.\ n\ f\ (f\ a))\ 0$

→ { β-reduction }

$\lambda f.\lambda a.\ 0\ f\ (f\ a)$

= { Definition of 0 }

$\lambda f.\lambda a.\ (\lambda f.\lambda a.\ a)\ f\ (f\ a)$

→ { β-reduction }

$\lambda f.\, \lambda a.\, (\lambda a.\, a)\ (f\ a)$

$\rightarrow$ { β-reduction }

$\lambda f.\, \lambda a.\, f\ a$

$=$ { Definition of 1 }

1

Notes

- A few other primitives (such as addition) can be defined in a similar way to SUCC;
- The predecessor function PRED is harder to define, but is possible too;

Other functions (such as multiplication and exponentiation) require a treatment of recursion in the λ-calculus.

RECURSION

In Haskell, we can define functions by using recursion. For example:

```
fac = \n -> if n == 0 then
                1
            else
                n * fac (n-1)
```

Question:

Is it possible to represent recursive functions in the λ-calculus, which has no built-in recursion mechanism?

Answer:

Amazingly, yes!

The solution involves fixpoints:

> A fixpoint of a function f is an expression x that is unchanged by applying f. That is, f x = x.

In turn:

> A fixpoint operator is a function fix that gives fixpoints, in that fix f is a fixpoint of any function f. That is, f (fix f) = fix f.

Let us suppose now that a fixpoint operator fix can be defined in the λ-calculus...

Then (using the Haskell syntax for numbers and Booleans), we define two functions:

```
fac' = λf. λn. if n==0 then
                   1
               else
                   n * f (n-1)

fac  = fix fac'
```

Notes:

- fac' and fac are non-recursive;
- fac' is obtained from the original fac definition by adding an extra parameter f to replace the use of recursion.

Finally, we calculate as follows:

```
  fac
=   { Definition of fac }
  fix fac'
=   { fix is a fixpoint operator }
  fac' (fix fac')
=   { Definition of fac' }
  (λf. λn. if n == 0 then
             1
           else
             n * f (n-1)) (fix fac')
:   { β-reduction }
```



```
  λn. if n==0 then
        1
      else
        n * (fix fac') (n-1)
=   { Definition of fac }
  λn. if n == 0 then
        1
      else
        n * fac (n-1)
```

That is,

> The non-recursive definition of fac using fix is equal to the recursive definition!

DEFINING A FIXPOINT OPERATOR

There are many ways to define a fixpoint operator in the λ-calculus.

The most famous one is called Y:

$$Y = \lambda f.\ \underbrace{(\lambda a.\ f\ (a\ a))}_{(*)}\ \underbrace{(\lambda a.\ f\ (a\ a))}_{(*)}$$

Notes:

- Y is non-recursive;
- Y comprises two identical subexpressions (*). Human DNA comprises two identical helices. Is this just a coincidence...?

EXERCISES

① Show that Y is a fixpoint operator.

② (Fun project.) Implement the operational semantics of the λ-calculus from lecture 11 in Haskell. Then try out some examples using the representations of Booleans etc. given in this lecture.

Principles of Programming Languages

Graham Hutton

Lecture 14 - Domain Theory I

INTRODUCTION

As we have seen, the lambda calculus can be given an operational semantics.

Question:

Can the lambda calculus also be given a denotational semantics?

Answer:

Yes - due to Dana Scott and Christopher Strachey's development of domain theory in the early 1970s.

The next three lectures introduce the basic principles of domain theory.

OUNDATIONAL PROBLEMS

ecall that a denotational semantics of language L is given by a pair

$$\langle D, [\![\ ]\!] \rangle$$

here

- D is a set of semantic values;
- $[\![\ ]\!] : L \to D$ is a valuation function, defined in a compositional manner.

nfortunately, there are two foundational problems ith using plain sets and functions.

① Recursively defined programs

Consider the Haskell programs

```
f :: Int → Int
g :: Int → Int
```

defined recursively by

```
f n = f n + 1        (*)
g n = g n            (**)
```

Operationally:

Evaluating `f n` or `g n` for any `n :: Int` will loop forever without giving a result.

But denotationally:

- No function $f : \mathbb{Z} \to \mathbb{Z}$ on the set $\mathbb{Z}$ of integers satisfies (*);
- Any function $g : \mathbb{Z} \to \mathbb{Z}$ satisfies (**).

Conclusion:

To properly deal with the denotational semantics of recursion (or iteration), we need an explicit notion of "looping" in the set of semantic values.

(2) Recursively defined semantic domains

For a denotational semantics of the lambda calculus, we require a set D such that

$$D \cong D \to D$$

where

- $S \to T$ is the set of (total) functions from set S to set T;
- $S \cong T$ means there is a bijection (1-to-1 correspondance) between S and T.

Problem:

The only solution is the trivial one $D = \mathbb{1}$, for any 1-element set $\mathbb{1}$.

As a simpler example, consider

$$D \cong D \to \mathbb{2}$$

or any 2-element set $\mathbb{2}$. Cantor's theorem states that $D \cong \mathbb{P}(D)$ has no solution.

since $\mathbb{P}(D) \cong D \to \mathbb{2}$, it follows that $D \cong D \to \mathbb{2}$ also has no solution.

Conclusion:

Higher-order languages lead to specifications for sets of semantic values that have no (non-trivial) solutions.

LIFTED SETS

As a first step to solving the problems, we assume each set of semantic values contains a special element $\perp$, such that

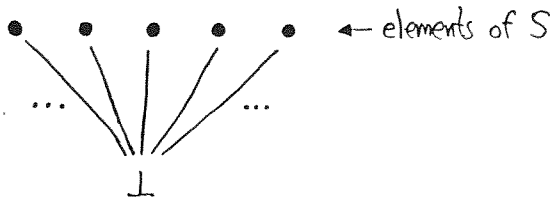
$\perp$ is pronounced "bottom";

$\perp$ represents { an undefined value; an error value; a looping computation.

Formally, given any set S:

The "lifted set" $S_\perp$ is the set $S \cup \{\perp\}$, where $\perp \notin S$.

Given the meaning of $\bot$, there is a natural "information content" ordering $\sqsubseteq$ on $S_\bot$:



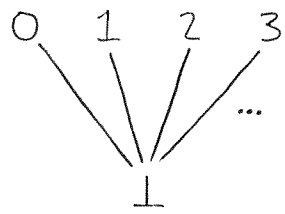
Formally, the ordering $\sqsubseteq$ is defined by:

$$x \sqsubseteq y \iff x = y \vee x = \bot.$$

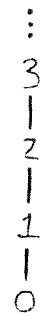
Notes

- To avoid confusion, we sometimes subscript $\bot$ and $\sqsubseteq$ with their underlying sets, as in $\bot_S$ or $\sqsubseteq_S$.

- Don't confuse the ordering $\sqsubseteq$ on $\mathbb{N}_\bot$:



with the standard ordering $\leqslant$ on $\mathbb{N}$:



Example: $0 \leqslant 1$, but $0 \not\sqsubseteq 1$ and $1 \not\sqsubseteq 0$, because from an information content point of view, 0 and 1 are incomparable.

EXAMPLE

Using lifted sets rather than plain sets solves our earlier problem with:

$$f\ n = f\ n + 1 \qquad (*)$$

Clearly, addition must satisfy $\bot + m = \bot$. That is, adding an undefined value to an integer gives an undefined value.

Now, $f\ n = \bot$ is the unique function $f : \mathbb{Z}_\bot \to \mathbb{Z}_\bot$ satisfying $(*)$, which expresses denotationally that $f\ n$ loops forever.

POINTED POSETS

Now consider the Haskell definition:

```
and      :: (Bool, Bool) -> Bool
and (b,c) = b && c
```

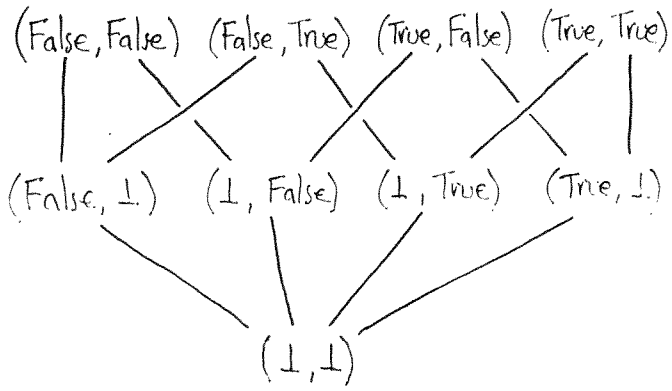
Clearly, the required denotation has type:

$$\mathbb{B}_\bot \times \mathbb{B}_\bot \to \mathbb{B}_\bot$$

where

- $\mathbb{B}$ is the set $\{False, True\}$ of Booleans;
- $S \times T = \{(x, y) \mid x \in S \wedge y \in T\}$ is the Cartesian product of sets S and T.

Observation: there is a natural information content ordering $\sqsubseteq$ on the set $\mathbb{B}_\perp \times \mathbb{B}_\perp$:



Intuition:

The information content of a pair of elements is increased by increasing the information content of either (or both) component of the pair.

Under this ordering, $\mathbb{B}_\perp \times \mathbb{B}_\perp$ is not itself a lifted set, but a "pointed poset".

Formally, a pointed poset is a triple

$$\langle S, \perp, \sqsubseteq \rangle$$

where

- S is a set;
- $\perp \in S$ is an element of S;
- $\sqsubseteq \subseteq S \times S$ is a relation on S, such that for all $x, y, z \in S$:
  - $\perp \sqsubseteq x$ (pointedness)

- $x \sqsubseteq x$ (reflexivity)
- $\dfrac{x \sqsubseteq y \wedge y \sqsubseteq x}{x = y}$ (antisymmetry)
- $\dfrac{x \sqsubseteq y \wedge y \sqsubseteq z}{x \sqsubseteq z}$ (transitivity)

Fact: if S and T are pointed posets, then so is the product $S \times T$, with:

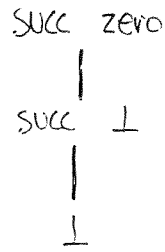
$$\perp_{S \times T} = (\perp_S, \perp_T)$$

$$(x, y) \sqsubseteq_{S \times T} (x', y') \Leftrightarrow x \sqsubseteq_S x' \wedge y \sqsubseteq_T y'$$

Fact: in denotational semantics, semantic values naturally form pointed cpos.

EXERCISES

① Given the pointed poset A, drawn as



draw the pointed poset $A \times A$

② Show that if S and T are both pointed posets, then so is $S \times T$.

③ Define $\perp_{S \to T}$ and $\sqsubseteq_{S \to T}$ such that $S \to T$ is also a pointed poset.

Principles of Programming Languages

Graham Hutton

Lecture 15 - Domain Theory II

INTRODUCTION

In the last lecture, we:

- Found that using sets of semantic values in denotational semantics caused some foundational problems with recursion;
- Postulated that semantic values naturally form pointed partially-ordered sets.

In this lecture, we:

- Postulate further structure on the road to giving a satisfactory denotational semantics to recursively defined programs.

MONOTONIC FUNCTIONS

If semantic values form pointed posets, then programs will naturally denote functions between pointed posets.

But ... not all functions are suitable, only those that preserve the information content orderings, in the sense that:

> Increasing the information content of the argument to a function increases the information content of its result.

Such functions are called "monotonic".

Formally, given pointed posets S and T, a function $f : S \to T$ is monotonic if

$$\forall x, y \in S.\ x \sqsubseteq_S y \Rightarrow f(x) \sqsubseteq_T f(y).$$

Note:

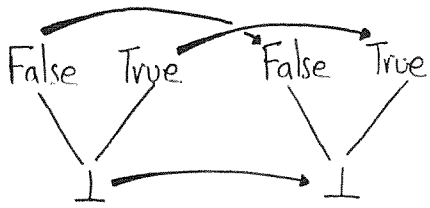
Monotonic functions preserve the $\sqsubseteq$ orderings, but are not required to preserve $\bot$. Functions for which $f(\bot) = \bot$ are called "strict" functions.

Fact:

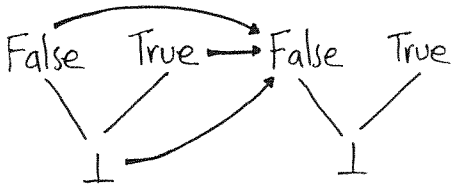
In denotational semantics, programs naturally denote monotonic functions.

Examples:

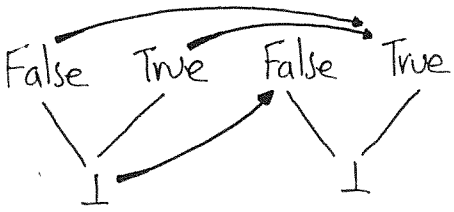
$f : \mathbb{B}_\perp \to \mathbb{B}_\perp$ — Monotonic?



✔ (identity function)



✔ (constant function)



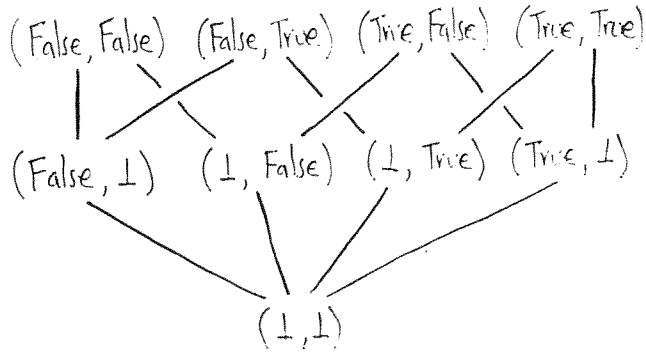
✘ ($\perp \sqsubseteq$ False, but $f(\perp) \not\sqsubseteq F(\text{False})$)

COMBINING INFORMATION

Two elements of a pointed poset can sometimes be "unioned" to combine the information content of the elements.

Example:

Consider the pointed poset $\mathbb{B}_\perp \times \mathbb{B}_\perp$:



Then,

$$(\text{False}, \perp) \sqcup (\perp, \text{True}) = (\text{False}, \text{True})$$

because:

(False, True) is the least element of the poset that subsumes both (False, ⊥) and (⊥, True) under the $\sqsubseteq$ ordering.

Similarly:

$$(\perp, \text{False}) \sqcup (\text{True}, \perp) = (\text{True}, \text{False})$$

$$(\text{False}, \perp) \sqcup (\perp, \perp) = (\text{False}, \perp)$$

$$(\text{False}, \perp) \sqcup (\text{False}, \text{True}) = (\text{False}, \text{True})$$

In general, however, not all elements of a pointed poset can be combined using ⊔.

Example: in $\mathbb{B}_\perp \times \mathbb{B}_\perp$,

$$(\text{False}, \perp) \sqcup (\text{True}, \perp)$$

does not exist, because

There is no (least) element of the poset that subsumes both elements.

Similarly,

$(\perp, \text{False}) \sqcup (\perp, \text{True})$ does not exist;

$(\text{False}, \text{False}) \sqcup (\text{False}, \text{True})$ does not exist.

LEAST UPPER BOUND

Formally, given a pointed poset S, the "least upper bound" (or lub) of two elements $x, y \in S$ is uniquely characterised (if it exists) by the following two properties:

① $x \sqsubseteq x \sqcup y \;\wedge\; y \sqsubseteq x \sqcup y$

"The lub subsumes both x and y"

② $\forall z \in S.\; x \sqsubseteq z \wedge y \sqsubseteq z \Rightarrow x \sqcup y \sqsubseteq z$

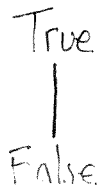
"The lub is the least element that subsumes both x and y".

We can also extend $\sqcup$ to sets of elements (rather than just two elements) in the obvious way, with $\bigsqcup \emptyset = \bot$ as the base-case.

Note

The abstract notion of a least upper bound operator $\sqcup$ generalises many familiar concrete operators.

Example: consider the simple poset



In this case, $x \sqcup y$ always exists, and is defined by the following equations:

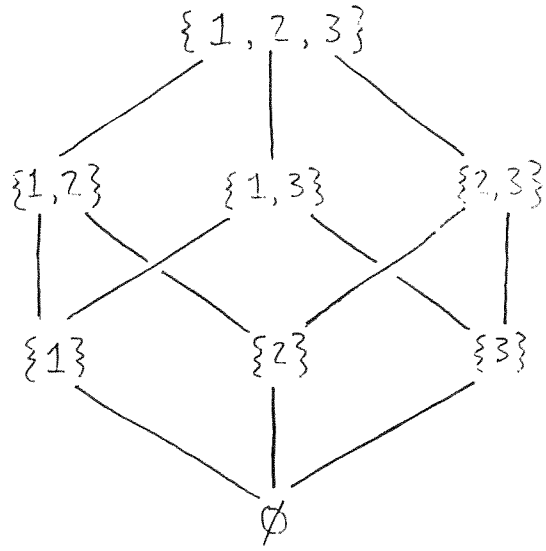
$$\text{False} \sqcup \text{False} = \text{False}$$
$$\text{False} \sqcup \text{True} = \text{True}$$
$$\text{True} \sqcup \text{False} = \text{True}$$
$$\text{True} \sqcup \text{True} = \text{True}$$

That is, $\sqcup$ is the "logical or" operator.

Example: for any set S, the powerset $\mathcal{P}(S)$ forms a pointed poset, with $\bot = \emptyset$ and $\sqsubseteq$ given by set inclusion $\subseteq$.

In this case, $\bigsqcup X$ always exists, and is given by set union $\bigcup X$.

For example, consider $\mathcal{P}\{1,2,3\}$:



RECURSION AND FIXPOINTS

As we saw in lecture 13, recursively defined programs can be expressed as fixpoints of non-recursively defined programs.

Example:

$$fac' = \lambda f.\lambda n.\ \text{if } n == 0 \text{ then } 1 \text{ else } n * f(n-1)$$

$$fac = fix\ fac'$$

where fix is a fixpoint operator.

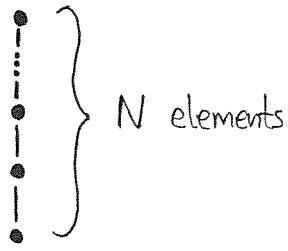
Hence, giving a denotational semantics to recursively defined programs can be reduced to the simpler problem of giving a semantics to the fix operator.

But... there are a few problems:

1. Some monotonic functions on pointed posets do not have a fixpoint;
2. Some such functions have many fixpoints - which one should we choose?
3. Is there an effective way to compute the desired fixpoint?

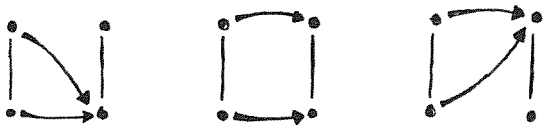
EXERCISES

1) Let $\mathbb{N}$ be the pointed poset:



There is one monotonic function $\mathbb{1} \to \mathbb{1}$:

There are three monotonic functions $\mathbb{2} \to \mathbb{2}$:



a) Write down the monotonic functions $\mathbb{3} \to \mathbb{3}$

b) Write a Haskell program to calculate the number of monotonic functions $\mathbb{N} \to \mathbb{N}$ for an arbitrary N.

2) Give an example of a monotonic function $f : S \to S$ on a pointed poset S that does not have a fixpoint.

Hint: S must be infinite.

3) Give a few other concrete examples of a least upper bound operator $\sqcup$.

Principles of Programming Languages

Graham Hutton

Lecture 16 - Domain Theory III

INTRODUCTION

In the last lecture, we postulated that:

- Semantic values form pointed posets;
- Programs denote monotonic functions.

In this lecture, we find that to give a denotational semantics to recursive programs, these postulates must be strengthened to:

- Semantic values form complete posets;
- Programs denote continuous functions.

CHAINS

To a first step to complete posets, we introduce the simple notion of a "chain".

Informally, a chain within a poset is a set of totally ordered elements:

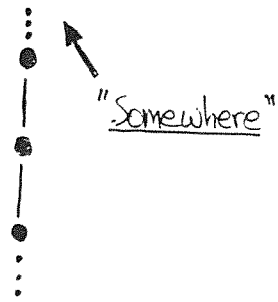
Note: a chain can be infinite in either (or both) directions.

Formally, a chain is a subset $T \subseteq S$ of poset S, such that:

$$\forall x, y \in T.\ x \sqsubseteq_S y \vee y \sqsubseteq_S x.$$

COMPLETE POSETS

It is natural to think of a chain as a set of elements that is "going somewhere":



In a complete poset, the chains "get there", in the sense that:

> Every chain has a least upper bound within the poset (but not necessarily within the chain itself.)

Formally, a complete poset (or cpo) is a poset S such that:

The least upper bound $\bigsqcup T$ exists within S for all chains $T \subseteq S$.

Note:

Since the empty set $\emptyset$ is a trivial chain, and $\bigsqcup \emptyset = \bot$, every cpo has a $\bot$, and is hence a pointed poset.

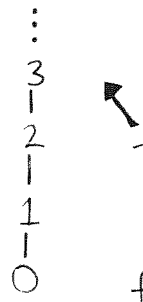
Fact:

In denotational semantics, semantic values naturally form complete posets.

Example: every finite pointed poset is a cpo.

Any (non-empty) finite chain has a greatest element, which is by definition the least upper bound for the chain.

Example: the infinite pointed poset of natural numbers ordered by $\leqslant$ is not a cpo:

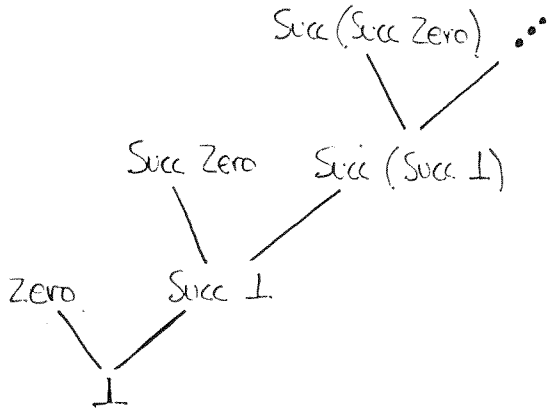


There is no greatest natural number, and hence no lub for the chain of all naturals.

Example: the Haskell datatype

```
data Nat = Zero | Succ Nat
```

forms a cpo, as follows:



In this case, the lub of the infinite chain $\bot \sqsubseteq \text{Succ}\ \bot \sqsubseteq \text{Succ}\ (\text{Succ}\ \bot) \sqsubseteq \ldots$ is the infinite value defined by $\text{inf} = \text{Succ}\ \text{inf}$.

CONTINUOUS FUNCTIONS

If semantic values form cpos, then programs will naturally denote functions between cpos.

But ... not all functions are suitable, only those that preserve the underlying structure of cpos, namely:

1. The partial ordering $\sqsubseteq$ on elements;
2. The least upper bound $\bigsqcup$ of chains.

Such functions are called "continuous".

Formally, given cpos S and T, a function $f : S \to T$ is continuous if:

① f is monotonic;

② $f(\bigsqcup X) = \bigsqcup \{f\,x \mid x \in X\}$ for all non-empty chains $X \subseteq S$.

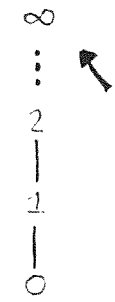
Note:

Using non-empty chains in ② ensures that continuous functions need not be strict.

Fact:

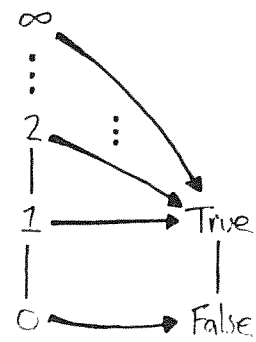
In denotational semantics, programs naturally denote continuous functions.

Example: let $\mathbb{N}^\infty$ be the cpo formed by adding infinity to the naturals ordered by $\leq$:



Infinity ∞ is the lub for the chain of all naturals.

Then, a continuous $f : \mathbb{N}^\infty \to 2$ is defined by:



$$f(x) = \begin{cases} \text{False}, & x = 0 \\ \text{True}, & x \neq 0 \end{cases}$$

THE FIXPOINT THEOREM

Recall: giving a denotational semantics to recursively defined programs reduces to giving a semantics to the fix operator.

This is given by the "fixpoint theorem":

> Every continuous function $f : S \to S$ on a cpo S has a "least fixpoint" $\mathrm{fix}(f)$, which can be computed as the lub of the infinite chain $\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \cdots$.
>
> Formally:
>
> $$\mathrm{fix}(f) = \bigsqcup \{f^n(\bot) \mid n \in \mathbb{N}\}$$

Note: the notation $f^n(\bot)$ means

$$\underbrace{f\,(f\,(f \cdots (\bot) \cdots)}_{\text{n-times}}$$

and is formally defined by:

$$f^0(x) = x$$
$$f^{n+1}(x) = f\,(f^n(x))$$

Note: the fact that $\mathrm{fix}(f)$ is the least fixpoint of f combines two properties:

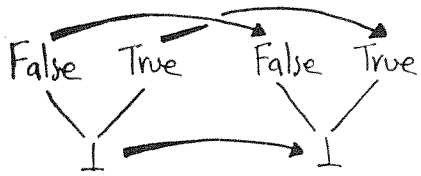
① $f(\mathrm{fix}(f)) = \mathrm{fix}(f)$

"$\mathrm{fix}(f)$ is a fixpoint of f";

② $\forall x \in S.\ f(x) = x \Rightarrow \mathrm{fix}(f) \sqsubseteq_S x$

"$\mathrm{fix}(f)$ is the least fixpoint of f"

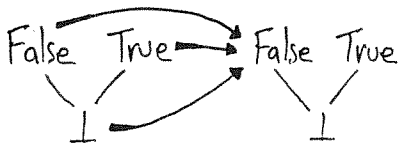
xample : given id : $\mathbb{B}_\perp \to \mathbb{B}_\perp$ defined by:



, we have:

$$\begin{aligned} fix(id) &= \bigsqcup \{ id^n(\perp) \mid n \in \mathbb{N} \} \\ &= \bigsqcup \{ \perp, id(\perp), id(id(\perp)), \ldots \} \\ &= \bigsqcup \{ \perp, \perp, \perp, \ldots \} \\ &= \perp. \end{aligned}$$

xample : given $f : \mathbb{B}_\perp \to \mathbb{B}_\perp$ defined by



, we have:

$$\begin{aligned} fix(f) &= \bigsqcup \{ \perp, f(\perp), f(f(\perp)), \ldots \} \\ &= \bigsqcup \{ \perp, \text{False}, \text{False}, \ldots \} \\ &= \text{False}. \end{aligned}$$

Example : given the Haskell datatype

```
data Nat = Zero | Succ Zero
```

the recursively definition for infinity:

```
inf = Succ inf
```

can be expressed as a fixpoint by:

```
inf  = fix inf'
inf' = λn → Succ n
```

Because Nat denotes a cpo (see slide 16.6), and inf' can be verified as denoting a continuous function on this cpo, the fixpoint theorem can be used to give the denotation of inf : ...

inf

= { definition of inf }

fix inf'

= { fixpoint theorem }

$\bigsqcup \{ \perp, inf'(\perp), inf'(inf'(\perp)), \ldots \}$

: { definition of inf' }

$\bigsqcup \{ \perp, Succ\ \perp, Succ\ (Succ\ \perp), \ldots \}$

: { property of Nat }

Succ (Succ (Succ (...

hat is, the fixpoint theorem gives the pected denotation for inf !

EXERCISES

① Modify the example on slide 16.9 to give an example of a function that is monotonic, but not continuous.

② Use the fixpoint theorem to calculate the denotation of the factorial function:

```
fac = λn → if n == 0 then
             1
           else
             n * fac (n-1)
```