Semantics of Computation

Lecture 1 — Introduction

References

A selection of textbooks on semantics, appropriate to this lecture course.


All three are in the MIT Press *Foundations of Computing* series. Winskel covers pretty much everything in the course, and at the right level. Gunter is rather more advanced, but the introductory chapter is well worth reading for its overview of what semantics is about. Mitchell is large, recent and comprehensive.

The following is an earlier article by Gunter on the scope and nature of semantics.


Two recent courses given at the University of Cambridge Computer Laboratory cover operational and denotational semantics in some depth. Conveniently, the lecture notes can be obtained electronically.


The formal definition of the programming language Standard ML provides a substantial example of applied semantics. It is set out in the following two books.


The first of these contains the rules that specify the semantics of the language (around 200 of them), while the second, complementary, text discusses various design decisions and their implications.

Attachments

Slides from lecture; copy of (Gunter 1992, pp. 1–9); copy of pages 52 and 53 of *The Definition of Standard ML*. This last sheet shows one third of the rules that define the dynamic semantics of the core language (i.e. excluding the module system).
Some aspects of programming language description

Syntax Characters, keywords and grammars for well-formed programs. Used to direct lexers and parsers.

✓ Semantics The meaning of data, expressions and program phrases: what they should do and how they interact.

Support environment Compilers, interpreters, standard libraries, debuggers etc. Examples of what the language is for and descriptions of how to use it.

What can semantics do for me?

Consider the following ML program:

```ml
fun ten f = let val c = ref 10
           in
               while !c > 0 do (dec c; f())
           end.
```

When is an application of `ten` certain to finish? Can `f` ever access cell `c` and does it matter? Is it safe to unroll the loop?

One task of a formal semantics is to provide a setting in which such questions make sense; and then supply the tools to answer them.
Areas of application

Language definition  A mathematical semantics can help avoid ambiguity and ensure consistency of implementation (cf. The Definition of Standard ML).

Reasoning about programs  Provides a basis for program logics and formal verification methods. Checks correctness of compiler analyses and code transformations.

Language design  Promotes exploration of the design space. Highlights the interaction between language features and draws attention to possible conflicts. May suggest new programming paradigms.

Varieties of semantics

Axiomatic  “Laws of programming” — equations and rules for reasoning about program phrases.

Operational  Specifies the computation steps a program phrase undertakes as it executes.

Denotational  Program phrases and language constructors are interpreted as elements of some abstract mathematical structure.

Also static vs. dynamic semantics.

Varieties of languages

We shall consider PCF, a simple, typed, functional language. Full-scale languages in this declarative style include Haskell, Lisp, Miranda™, Objective Caml and Standard ML.

In the exercises you will investigate a small imperative / procedural language.

There is active research in semantics for most other styles of language and aspects of computation:

- objects, concurrency, logic programs, constraint programs . . .
- hybrid systems, interaction, performance analysis, . . .
Compositionality

A semantics is *compositional* if the behaviour of a program phrase can be consistently derived from the behaviour of its parts.

This is the crucial lever that allows us to tackle large problems, by breaking them down into smaller ones.

Furthermore, it gives assurance that small well-understood systems can be soundly combined to build larger ones.

Overview of the course

<table>
<thead>
<tr>
<th>Topic</th>
<th>Mon.</th>
<th>Fri.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Induction</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>Operational Semantics</td>
<td>15</td>
<td>19</td>
</tr>
<tr>
<td>Domain Theory</td>
<td>22</td>
<td>26</td>
</tr>
<tr>
<td>Denotational Semantics</td>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>17</td>
</tr>
</tbody>
</table>

Literature

*Winskel*  *The Formal Semantics of Programming Languages.*

*Gunter*  *Semantics of Programming Languages.*

*Mitchell*  *Foundations for Programming Languages.*

The following notes, available electronically, cover several topics of the present course in some detail.

*Pitts*  *Semantics of Programming Languages and Denotational Semantics.* Lecture notes for courses at the University of Cambridge Computer Laboratory.
References

Chapters 3 and 4 of (Winskel 1993) are all about induction and inductive definitions. This includes the more general notion of well-founded induction. Chapter 2 of (Pitts 1997a) is close to the approach we use.

The classic foundation for this material is the Knaster-Tarski fixed-point theorem, as applied to lattices of subsets.


I believe that structural induction was introduced to computer science by Burstall.


The language PCF was derived by Plotkin from Scott’s LCF, a Logic for Computable Functions.

G. D. Plotkin. LCF considered as a programming language. Theoretical Computer Science 5:223–255. 1977

This was also the paper that introduced the notion of a ‘fully abstract’ model for a programming language, and described the corresponding rôle of ‘parallel or’, which we shall encounter later in the course.

Attachments

Slides from lecture.
Mathematical induction

We take \( \mathbb{N} = \{0, 1, 2, \ldots\} \) as the natural numbers. Suppose that \( \phi(x) \), where \( x \in \mathbb{N} \), is some property for which we wish to show
\[
\forall x \in \mathbb{N} . \phi(x) .
\]
Then it is enough to prove that
\[
\phi(0) \quad \& \quad \forall x \in \mathbb{N} . \phi(x) \Rightarrow \phi(x + 1)
\]
or more generally
\[
\forall x \in \mathbb{N} . (\forall y < x . \phi(y)) \Rightarrow \phi(x) .
\]

Finite labelled trees

When we write syntax like
\[
\text{while \( !c > 0 \) do \( \text{dec} \ c; f() \)}
\]
we shall really mean something like

```
    while,do  \( \text{dec} \ c; f() \)
      \( !c \) \( 0 \)
```

a tree built from leaves by constructors. This exposes more clearly the structure within the phrase.

Structural induction

To prove that a property holds of all finite labelled trees, it is enough to show the following:

**base case:** for every kind of leaf \( L \), the property holds of the one-node tree \( L \); and

**induction step:** for every \( n \)-argument constructor \( C \), if the property holds for trees \( T_1, T_2, \ldots, T_n \) then it holds for \( C(T_1, T_2, \ldots, T_n) \).

This principle is in turn proved by mathematical induction on tree size.
Example: Binary trees

Consider trees of the form

\[ \text{BTree} = \text{Leaf} \mid \text{Fork(Btree, Btree)} \].

For every such tree \( T \) we show that \( 2 \times \text{width}(T) \leq 2^{\text{depth}(T)} \).

**Base case:** \( \text{width(Leaf)} = 1 \), \( \text{depth(Leaf)} = 1 \), \( 2 \times 1 \leq 2^1 \).

**Induction step:**

\[
2 \times \text{width(Fork}(T_1, T_2)) \leq 2 \cdot (2^{\text{depth}(T_1)} + 2^{\text{depth}(T_2)})
\]

\[
\leq 2 \cdot 2^{\max(\text{depth}(T_1), \text{depth}(T_2))}
\]

\[
= 2^{\text{depth}(\text{Fork}(T_1, T_2))}
\]

\[
\text{definition}
\]

Inductive definitions

Given a set \( T \), an **inductively defined subset** \( I \subseteq T \) is one given by a collection of axioms and rules:

**Axiom**

\[ \alpha \]

**Rule**

\[ \frac{h_1, h_2, \ldots, h_n}{c} \]

This axiom asserts that \( \alpha \) is in \( I \), while the rule states that if the hypotheses \( H = \{h_1, \ldots, h_n\} \) are all in \( I \) then so is the conclusion \( c \).

The subset \( I \) is to be the least containing all the axioms and closed under all the rules.

Example: Divisibility

Take the set \( \mathbb{N} \times \mathbb{N} \) and consider the following rules:

\[
p \mid 0 \quad p \mid p \quad p \in \mathbb{N}
\]

\[
p \mid q \quad p \mid r \quad p, q, r \in \mathbb{N}
\]

These inductively define a subset \( \text{Div} \subseteq \mathbb{N} \times \mathbb{N} \) containing \( (3, 3), (2, 8), (5, 0), \ldots \).
Rule induction

Suppose that the subset $I \subseteq T$ is inductively defined by axioms $A$ and rules $R$. In order to prove for some property $\phi$ that

$$\forall t \in I . \phi(t)$$

it is enough to show the following:

**base case:** $\phi(a)$ holds for each axiom $a \in A$; and

**induction step:** for each rule $(H, c) \in R$, if $\phi(h)$ holds for every hypothesis $h \in H$ then $\phi(c)$ also holds for the conclusion.

---

Example: Divisibility means divisibility

We expect that the following will hold:

$$p \mid q \implies \exists i \in \mathbb{N}. q = p \times i.$$  

Call the right hand side $\phi(p, q)$ and proceed by rule induction:

**base case:** $\phi(p, 0)$ as $0 = p \times 0$ and $\phi(p, p)$ as $p = p \times 1$;

**induction step:** given $\phi(p, q)$ and $\phi(p, r)$, with $q = p \times i$ and $r = p \times j$, then $\phi(p, q + r)$ holds as $q + r = p \times (i + j)$.

Thus $\phi(p, q)$ for every $(p, q)$ in $\text{Div}$, as required.

---

PCF — Programming language for Computable Functions

**Types**

$$\sigma ::= a \mid i \mid \sigma \rightarrow \sigma'$$

**Terms**

$$M ::= x^\sigma \mid MN \mid \lambda x^\sigma . M \mid tt \mid ff \mid if \: i_{\sigma} \mid if \: i_i$$

$$\mid n \mid iszero \mid succ \mid pred \mid Yo$$

**Free variables**

$$fv(x) = \{x\} \quad fv(MN) = fv(M) \cup fv(N)$$

$$fv(c) = \emptyset \quad fv(\lambda x . M) = fv(M) \setminus \{x\}$$
Substitution and $\alpha$-conversion

We identify PCF terms up to relabelling of bound variables ($\alpha$-conversion). For example

$$\lambda x.(\text{succ } x) \quad \text{and} \quad \lambda y.(\text{succ } y)$$

are considered to be the same term.

The substitution $M[N/x]$ replaces all free occurrences of $x$ in $M$ with $N$, $\alpha$-converting where necessary to avoid capturing the free variables of $N$.

We write simultaneous substitution as $M[N_1/x_1, \ldots, N_k/x_k]$.

Type assignment

Type judgements $M : \sigma$ are elements of the inductively defined subset $J \subseteq \text{Terms} \times \text{Types}$ given by the following axioms and rules:

- $x^\sigma : \sigma$
- $M : \sigma \to \sigma' \quad N : \sigma \quad \overline{MN : \sigma'}$
- $\lambda x^\sigma . M : \sigma \to \sigma'$
- $tt : o$
- $zero : i$
- $if_0 : o \to o \to o$
- $ff : o$
- $succ : i \to i$
- $if_1 : o \to i \to i$
- $n : i$
- $pred : i \to i$
- $if_2 : o \to i \to i$
- $Y_o : (\sigma \to \sigma) \to \sigma$

We consider only well-typed terms $M$, i.e. such that $M : \sigma \in J$ for some type $\sigma$. 
Semantics of Computation

Lecture 3 — Operational Semantics I

References

The classic SOS source is Plotkin’s DIAMI lecture notes. Get a copy from the information office.


Natural semantics was developed by Kahn and his research group at INRIA.


Standard ML is defined through a formal operational semantics of this kind, but the techniques are not limited to modern functional languages. The following research paper describes their use in a quite different context.


This successfully handles for example the non-determinism inherent in ISO Standard C. The semantics is built within HOL, a mechanical theorem prover, which then allows automatic reasoning about program behaviour.

Some references to other rule-based systems for operational semantics. The first describes the GSOS system, the second the so-called ‘de Simone format’.


Numerous textbooks and research papers describe abstract machines of one kind or another. Plotkin (1981, §1.5.2) presents the SMC machine for a small imperative language; this is reproduced by Pitts (1997a, §1.2). Gunter (1992, pp.23–26) describes Landin’s SECD machine for a simple functional language.

Attachments

Slides from lecture; exercise sheet A; description of the SECD machine from (Gunter 1992); copy of the article LCF considered as a programming language (Plotkin 1997).
Reduction semantics for PCF

An inductively defined set $\rightarrow \subseteq \text{Terms} \times \text{Terms}$ of reductions.

$$
\begin{align*}
M & \rightarrow M' & B & \rightarrow B' & N & \rightarrow N' \\
MN & \rightarrow M'N & \text{if } B & \rightarrow B' & \text{if } BM & \rightarrow BM' \\
\text{(}\lambda x. M\text{)}N & \rightarrow M[N/x] & \text{if } \text{tt } M & \rightarrow M' & \text{succ } n & \rightarrow m \quad (m = n + 1) \\
YM & \rightarrow M(YM) & \text{if } \text{ff } MM & \rightarrow M' & \text{pred } m & \rightarrow n \quad (m = n + 1) \\
iszero 0 & \rightarrow \text{tt} & \text{iszero } n & \rightarrow \text{ff } (n > 0) & \text{pred } 0 & \rightarrow 0
\end{align*}
$$

Also called small-step or transition semantics. Function application here is call-by-name.

More on reduction

Some terminology associated with reduction semantics.

- $M \rightarrow^+ M'$ is the transitive closure of $\rightarrow$.
- $M \rightarrow^* M'$ is the transitive-reflexive closure of $\rightarrow$.
- We write $M \rightarrow$ if $M \rightarrow M'$ for some $M'$, otherwise $M \not\rightarrow^*$.
- A term converges $M\uparrow$ if $M \rightarrow^* V \not\rightarrow^*$ for some $V$.
- A term diverges $M\uparrow$ if it does not converge; equivalently if there is some unending reduction sequence

$$
M \rightarrow M_1 \rightarrow M_2 \rightarrow M_3 \rightarrow \ldots
$$

Values and evaluation

The values of PCF are a subset $\text{Values} \subseteq \text{Terms}$ comprising

- constants: if, n, iszero, succ, pred, Y;
- partial applications: if B, if BM;
- function abstraction: $\lambda x^\beta. M$.

An evaluation semantics is a relation $\Downarrow \subseteq \text{Terms} \times \text{Values}$.

Also called big-step, natural or relational semantics.
Evaluation semantics for PCF

\[
\begin{align*}
V & \Downarrow_V & M & \Downarrow_V & VN & \Downarrow_V & M & \Downarrow_N & n & \text{(} m = n + 1 \text{)} \\
M[N/x^n] & \Downarrow_V & M & \Downarrow_V & \text{if } B M M' & \Downarrow_V & M & \Downarrow_{\text{pred}} & M & \Downarrow_V & n & \text{(} m = n + 1 \text{)} \\
\lambda x^M N & \Downarrow_V & B & \Downarrow_{\text{tt}} & M & \Downarrow_V & M & \Downarrow_{\text{pred}} & M & \Downarrow_V & n & \text{if } B M' & \Downarrow_V & M & \Downarrow_{\text{pred}} & M & \Downarrow_V & 0 \text{ if } (R[-]) & M & \Downarrow_V & c (R[-]) & (n > 0)
\end{align*}
\]

We write \( M \Downarrow \) if \( M \Downarrow V \) for some \( V \), and \( M \uparrow \) if not.

Structural Operational Semantics (Plotkin 1981)

The motivation for SOS is that semantics should formalize intuition rather than implementation.

- Operational steps should all have computational content and be of the ‘right size’.
- Semantics should be syntax-directed, matching the structure of the language.
- There should be a minimum of non-language ‘scaffolding’, with configurations of the system being simple and comprehensible.

All this is helped by the innovation of attaching proofs to transitions.

Reduction contexts

In an axiom \( K \rightarrow K' \) of the reduction semantics, \( K \) is the \textit{redex} and \( K' \) the \textit{reduct}. For example \( (\lambda x^M) N \) is a \textit{\( \beta \)-redex}.

A \textit{reduction context} is a certain kind of term with a hole in it:

\[
R[-] ::= [-] \mid [R[-)]N \mid \text{if } (R[-]) M M' \mid c (R[-])
\]

These identify the site of immediate computation within a term.

Every PCF term is either a value or decomposes uniquely as a redex in a reduction context. This then determines its next reduction step:

\[
M = R[K] \rightarrow R[K']
\]

If any context is a reduction context then we have a \textit{rewrite system}. 

3-4

3-5

3-6
Labelled transition systems

Given a set of States and Labels, a labelled transition system is a subset $T \subseteq \text{States} \times \text{Labels} \times \text{States}$. Elements are written as $s \xrightarrow{a} s'$. Some states may be highlighted as initial or terminal.

An LTS can be specified explicitly; by inductive rules; or by other rule formats (GSOS, de Simone, tyft/tyxt, ...).

$$
\frac{p \xrightarrow{a} p'}{p \parallel q \xrightarrow{a} p' \parallel q} \quad \text{and} \quad \frac{q \not\xrightarrow{a} q'}{p \parallel q \not\xrightarrow{a} p \parallel q'}
$$

LTS's also link to the broad subject of automata, Petri nets and other concurrent systems.

3-7

Abstract machines

One way to give a language semantics is to implement it on a hypothetical specialised processor that directly manipulates language terms through stacks, registers and so forth.

Examples include the SMC machine, the SECD machine, Krivine’s machine, the CAM, ...

These abstract machines are low level, with every step directly justified by semantic intuition. This is good for expressing techniques to interpret or compile high-level languages. They do not however capture the semantics itself, which makes them poor for reasoning about general properties of languages and programs.

3-8

Some varieties of operational semantics

Abstract machine

Reduction semantics  Evaluation semantics

3-9
References

Winskel (1993, §10.1) explains Bekić’s Theorem which underlies the definition on slide 4-3 of mutual recursion in terms of simple recursion.

Proofs of the equivalence between reduction and evaluation semantics for a small imperative language can be found in (Pitts 1997a, §3) and (Winskel 1993, Ex. 4.10, p. 51).

Pitts (1997a, §8) outlines operational uses of coinduction for treating divergence and bisimulation. Some other references are as follows: the last article covers a range of the latest results in operational reasoning.

A. M. Pitts. Some notes on inductive and co-inductive techniques in the semantics of functional programs (draft version). BRICS Notes Series NS-94-5, Department of Computer Science, University of Aarhus, December 1994.


The following is the original source for the context lemma.


Description and proof of the ‘ciu’ theorem can be found in the following.


Logical relations, originally a purely denotational notion, now appear in many areas of semantics. These two papers provide examples of their operational use for languages with store.


**Attachments**

Slides from lecture; exercise sheet B.
The Y combinator

We have the reduction rule

\[ Y \alpha M \rightarrow M(Y \alpha M) \]

where \( M \) has type \( \sigma \rightarrow \sigma \). When \( M \) is an abstraction this becomes

\[ Y(\lambda f. N) \rightarrow^* N[\{Y(\lambda f. N)\}/f] \]

which is enough to encode recursion.

```
double \( \overset{\text{def}}{=} Y(\lambda f. \text{if } (\text{iszero } x) 0 (\text{succ}(\text{succ}(\text{f}(\text{pred } x))))))
```

```
double n \rightarrow^* \text{succ}(\text{succ}(\text{double}(\text{pred } n))) \rightarrow^* 2n \quad (n > 0)
double 0 \rightarrow^* 0
```

Recursion by \( \mu \)

An alternative approach is to include the abstraction \( \mu x. M \) in the language:

\[
\begin{align*}
\frac{M : \sigma}{\mu x. M : \sigma} & \quad \frac{\mu x. M \rightarrow M[\mu x. M/x]}{M[\mu x. M/x] \Downarrow V} \\
\end{align*}
\]

The effect however is the same; if we define

\[
\tilde{Y} \overset{\text{def}}{=} \mu f. \lambda x. (fx) \quad \text{and} \quad \tilde{\mu} x \overset{\text{def}}{=} Y(\lambda x. M)
\]

then we can derive

\[
\tilde{Y} M \rightarrow^* M[\tilde{Y} M] \quad \text{and} \quad \tilde{\mu} x. M \rightarrow^* M[\tilde{\mu} x. M/x] .
\]

Recursive function definitions

One can conveniently express recursive functions by declaration:

\[
\text{fun } f \ x = M .
\]

This can be encoded as

\[
\mu f. \lambda x. M \quad \text{or as} \quad Y(\lambda f. \lambda x. M) .
\]

Mutual recursion is also possible:

\[
\text{fun } f \ x = M \quad \mu f. \lambda x. (\lambda g. M)(\mu g. \lambda y. N) \\
\text{and } g \ y = N \quad \mu g. \lambda y. (\lambda f. N)(\mu f. \lambda x. M) .
\]

Finally, declaring \( \text{fun } Y x = x(Y x) \) takes us back to where we started.
Properties of reduction

Reduction of closed terms preserves types

\[ M : \sigma \text{ & } M \rightarrow M' \implies M' : \sigma \]

and is deterministic

\[ M \rightarrow M' \text{ & } M \rightarrow M'' \implies M' = M''. \]

Both can be proved by rule induction on \( M \rightarrow M' \). For example, the first one involves showing that the property

\[ \phi(M, M') \overset{\text{def}}{=} (\forall \sigma. M : \sigma \Rightarrow M' : \sigma) \]

is preserved by the rules defining \( \rightarrow \).

Exactly the same properties hold for evaluation \( M \Downarrow V \).

Equivalence of reduction and evaluation

For any closed term \( M \) and value \( V \):

\[ M \Downarrow V \iff M \rightarrow^* V. \]

The proof comes in three parts:

(i) \( M \Downarrow V \implies M \rightarrow^* V \) (rule induction on \( M \Downarrow V \))

(ii) \( M \rightarrow M' \text{ & } M' \Downarrow V \implies M \Downarrow V \) (rule induction on \( M \rightarrow M' \))

(iii) \( M \rightarrow^* V \implies M \Downarrow V. \)

Part (iii) follows from (ii) by mathematical induction on the length of the reduction sequence.

Nontermination

Terms like \( Y(\lambda x.x) \) and \( \mu x.(suc x) \) diverge:

\[ Y(\lambda x.x) \rightarrow (\lambda x.x)[Y(\lambda x.x)] \rightarrow Y(\lambda x.x) \rightarrow \cdots \]

In proving this we use the least number principle to show that there is no finite terminating reduction sequence; or, equivalently, no finite proof of evaluation.

\[
\begin{align*}
\text{?} & \quad \mu x.(suc x) \Downarrow n \\
\text{suc} \mu x.(suc x) & \Downarrow m \quad \{m = n + 1\}.
\end{align*}
\]

With coinduction we can give rule-based definitions of \( M \uparrow \) and \( M \Downarrow \), and hence deduce coinductive reasoning principles.
Equivalence

Two closed PCF terms of the same type are Kleene equivalent if either they both diverge or they both evaluate to the same value.

\[ M_1 \overset{k1}{\equiv} M_2 \iff \forall V \in \text{Values} (M_1 \Downarrow V \iff M_2 \Downarrow V) . \]

For example:

- \( M \to M' \implies M \overset{k1}{=} M' \) if \( B M_1 M_2 \overset{k1}{=} \) if \( (\text{not } B) M_2 M_1 \)
- \( M \Downarrow V \implies M \overset{k1}{=} V \)
- \( Y \text{succ} \overset{k1}{=} Y (\lambda x. x) . \)

Equational reasoning

Reflexivity \( M = M \)

Symmetry \( M = M' \implies M' = M \)

Transitivity \( M = M' \implies M' = M'' \implies M = M'' \)

Congruence \( C[M] = C[M'] \)

When a relation is closed under these rules we can use it for the familiar process of equational reasoning.

Problems

Kleene equivalence permits equational reasoning, but makes ‘too many’ distinctions between terms.

- \( \lambda x. (\text{not } ff) \not\overset{k1}{=} \lambda x. tt \)
- \( \lambda x. (\text{pred} (\text{succ} x)) \not\overset{k1}{=} \lambda x. x \)
- \( \lambda x. (\lambda y. y) x \not\overset{k1}{=} \lambda x. x \)
- \( M \not\overset{k1}{=} \lambda x. M x \)

The issue here is that some functions with different textual forms may nevertheless be indistinguishable when in use.
We wish to determine when two terms can be exchanged in any program without affecting its observed behaviour.

Define a program $P$ to be a closed term of ground type (o or i) with behaviour $\text{Beh}(P) \in \{\emptyset, \text{tt}, \text{ff}, 0, 1, 2, \ldots\}$ according to its evaluation.

A context $\mathcal{C}[-]$ is a program with one or more occurrences of a typed hole ‘$-$’. Define

$$M_1 \equiv_{ctx} M_2 \iff \forall \mathcal{C}[-]. \text{Beh}(\mathcal{C}[M_1]) = \text{Beh}(\mathcal{C}[M_2]).$$

We say that these terms are contextually equivalent.

---

Because it considers all possible programs, contextual equivalence is

- the right notion for checking code transformation, replacing algorithms, checking assertions and matching specifications;
- hard to demonstrate in any particular case.

Thus we look for either a simpler characterisation of contextual equivalence, or other relations that imply it but are simpler to demonstrate. (Kleene equivalence is one)

---

In order to prove contextual equivalence it is sufficient to consider program contexts of the form

$$[-]V_1V_2\ldots V_n.$$

This is an extensionality principle for PCF: the observable behaviour of function terms is limited to their action as maps from arguments to results.

Proof is by detailed analysis of reduction sequences $\mathcal{C}[M] \to^* V$ or through an alternative characterisation of $\equiv_{ctx}$ using logical relations.
Other relations contained in $\equiv_{\text{ctx}}$

The context lemma leads to a notion of **applicative equivalence**, defined by induction on types.

In languages other than PCF, particularly ones with *side-effects*, the context lemma may fail. Often though a *ciu* theorem is available, saying that reduction contexts $R[\_\_\_]$ characterise contextual equivalence.

Sometimes **logical relations** are useful, or coinductive notions like **applicative bisimulation**.

All of these relations generalise to open terms, and most of them can also be refined to preorders $\preceq_{\text{ctx}}$. 

---

4-13
Semantics of Computation

Lecture 5 — Operational Semantics III

References

The following papers address the distinction between call-by-name, call-by-value and call-by-need PCF. Note that Plotkin (1981, pp. 164–165) discusses six different strategies for function application.


Both Winskel (1993) and Pitts (1997a) give operational semantics for storage, in the context of small languages of commands. For the interaction between store and functions see (Pitts & Stark 1997) and the references therein. On functions and I/O:


The *Definition of Standard ML* includes structural operational semantics for the first four features listed on slide 5-6. Mosses work on action semantics also aims to provide a common basis for several of these ‘facets’ of computation. Following this, a reference for those who can’t live without OO programming.


Two standard reference texts on calculi for communication.


Attachments

Slides from lecture; exercise sheet C; extract from *The IV Language Reference Manual*; copy of the article *Call-by-name, call-by-value and the \( \lambda \)-calculus* (Plotkin 1975).
Call by name — Call by value

Function application in PCF is call-by-name or lazy:

\[ (\lambda x.M)N \rightarrow_n M[N/x] \]

An alternative VPCF has a call-by-value or strict semantics:

\[ N \rightarrow_v N' \quad (\lambda x.M)\rightarrow_v M[x] \]

Different Y rules.

Contextual equivalence is different in the two languages, though using continuation-passing style one can give translations between them.

---

Call by need

Real lazy functional languages like Haskell replace call-by-name with an equivalent but more efficient call-by-need semantics:

\[ (\lambda x.\mathbf{x} + (x + 3))\mathbf{N} \rightarrow \mathbf{N} + (\mathbf{N} + 3) \rightarrow \mathbf{N} + 5 + 8 \rightarrow 13 \]

Strict and lazy data

Consider adding pairs \( (M, N) : \sigma \times \sigma' \) to PCF. There is a choice of semantics. A strict version that evaluates inside the pair:

\[ M \downarrow V, N \downarrow W \quad \text{fst}(V, W) \downarrow V \quad \text{snd}(V, W) \downarrow W \]

and a lazy version where all pairs are values and evaluation is only carried out when required:

\[ (M, N) \downarrow (M, N) \quad M \downarrow V \quad N \downarrow W \]

The strict form of a list datatype allows only finite lists, while the lazy form includes potentially infinite streams.
State and storage

Consider extending VPCF with state in the form of a countably infinite supply of locations \( l \); a type \( \lambda \) for them; stores \( \sigma : \text{Locations} \rightarrow \mathbb{N} \) that are zero almost everywhere; and operations get : \( \lambda \rightarrow \tau \) and set : \( \lambda \rightarrow \tau \rightarrow \tau \). Existing operational rules need extra structure:

\[
\frac{\sigma, M \rightarrow \sigma', M'}{\sigma, MN \rightarrow \sigma', MN}
\]

and the new operations need rules of their own:

\[
\frac{\sigma, L \equiv \sigma', 1 \quad (n = \sigma'(1))}{\sigma, \text{get } L \equiv \sigma', n}
\]

\[
\frac{\sigma, L \equiv \sigma', 1 \quad \sigma', N \equiv \sigma''(n), n}{\sigma, \text{set } LN \equiv \sigma''(1 \rightarrow n), n}
\]

The left-to-right ordering of evaluation is now significant.

Input and output

Suppose we add simple I/O to VPCF, with operations read : \( \iota \rightarrow \tau \) and write : \( \iota \rightarrow \tau \). State is now captured by a pair of finite strings of numbers: input to be consumed \( \text{‘in’} \) and output produced \( \text{‘out’} \).

\[
\begin{align*}
\text{in } m \Rightarrow \text{read } & \Rightarrow \text{out } \Rightarrow \text{in } m \Rightarrow \text{out} \\
\text{in } m \Rightarrow \text{write } & \Rightarrow \text{out } \Rightarrow \text{in } n \Rightarrow \text{out } \\
\text{in } N \Rightarrow \text{out } & \Rightarrow \text{in } m \Rightarrow \text{out }' \\
\text{in } \Rightarrow \text{write } N & \Rightarrow \text{out } \Rightarrow \text{in } \Rightarrow \text{n } \Rightarrow \text{non } \Rightarrow \text{out }'
\end{align*}
\]

We need to reconsider observations and contextual equivalence. For example, it is now possible to have two divergent terms that are observably distinct.

And so on...

Structural operational semantics have been devised for a host of language features, including:

- recursive datatypes;
- dynamically allocated store;
- exceptions;
- module systems (ML structures and functors);
- coroutines and concurrency (CML);
- channel-based value communication;
- objects (c.f. Abadi and Cardelli).

A common methodology makes it possible to study not just individual features but also their interactions with one another.
We take an alphabet of names $a, b, c, \ldots$, which can be thought of as channels for synchronisation.

Language terms represent processes or agents

$$P ::= 0 | a.P | P + P' | P \mid P \backslash a$$

and are identified up to structural congruence

$$P + Q \equiv Q + P \quad P \equiv P + 0 \quad (P + Q) + R \equiv P + (Q + R)$$

$$P \mid Q \equiv Q \mid P \quad P \equiv P \mid 0 \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R).$$

Full CCS adds relabelling $P[f]$ and allows mutually recursive agent definitions.

**Operational semantics for CCS**

Agents may perform three kinds of action: input $a$, output $\overline{a}$ or a silent internal transition $\tau$.

$$a.P \xrightarrow{a} P \quad P \xrightarrow{0} P + Q \xrightarrow{a} P' \quad P \xrightarrow{\overline{a}} Q \xrightarrow{a} Q'$$

$$\overline{\pi}.P \xrightarrow{\pi} P \quad P \xrightarrow{\pi} Q \xrightarrow{a} Q' \quad P \mid Q \xrightarrow{a} P' \mid Q'$$

This system is nondeterministic in that a single agent may have several different transitions enabled.

Identifying agents up to structural congruence means that we do not need the symmetric duals of the $+$ and $\mid$ rules.

**Bisimilarity**

A symmetric relation $S \subseteq \text{Agents} \times \text{Agents}$ is a bisimulation if

$$\forall (P, Q) \in S . P \xrightarrow{a} P' \Rightarrow \exists Q'. Q \xrightarrow{\overline{a}} Q' \& (P', Q') \in S.$$ 

Two processes are bisimilar $P \sim Q$ if there is some bisimulation relating them: intuitively, if each can match every transition of the other. For example:

$$a \mid b \sim a.b + b.a$$

an instance of the expansion law;

$$a.(b + c) \not\sim a.b + a.c$$

although they are trace equivalent.
Properties of bisimilarity

Bisimilarity ‘ℓ’ is itself a bisimulation, the largest one. It is an equivalence relation and preserved by all the CCS operations, so can be used for equational reasoning. There is an expansion law:

\[
\left( \sum \alpha_i . P_i \right) \cup \left( \sum \beta_j . Q_j \right) = \sum \alpha_i . \left( P_i \cup \sum \beta_j . Q_j \right) + \sum \beta_j . \left( \sum \alpha_i . P_i \cup Q_j \right) + \sum_{(i,j) \in M} \tau . (P_i \cup Q_j)
\]

\[M = \{ (i,j) \mid \alpha_i = a \land \beta_j = \overline{a} \text{ or vice versa for some name } a \} .\]

More sophisticated reasoning methods include ‘bisimulation up to bisimilarity’, ‘bisimulation up to congruence’ and various techniques that make use of the coinductive nature of bisimilarity.

More on communicating systems

Milner’s CCS and Hoare’s CSP provide foundations for the semantics of communicating systems. Descendants of these calculi include:

- value-passing CCS where data may pass along channels;
- higher-order calculi which allow process-passing and more;
- the π-calculus which captures mobility of processes by passing names along channels to give dynamic reconfiguration;
- the join-calculus for distributed systems;
- the spi-calculus for reasoning about security protocols.
Semantics of Computation

Lecture 6 — Denotational Semantics and Domain Theory

References

All of the three textbooks mentioned in Lecture 1 cover domain theory and denotational semantics in fair detail (Winskel 1993, Gunter 1992, Mitchell 1996). Pitts’s lecture notes (1997b) provide a good overview, though they do not progress to recursively defined domains. Chapters 1–4 of the book below give an introduction to domains from a mathematical viewpoint.


You may be able to get hold of copies of the notes from my own *Domain Theory* course given in Spring 1997.

The following are worth consulting as references on the origins of denotational semantics and domain theory.


Scott’s influential article circulated for 24 years as a typewritten manuscript before finally being published!

For serious work, these two texts detail everything you could possibly want to know about domains.


http://hypatia.dcs.qmw.ac.uk/authors/J/JungA/papers/handy.ps.gz


http://hypatia.dcs.qmw.ac.uk/sites/other/domain.notes.other

Both are available from the electronic archive *Hypatia* at the addresses indicated.

Attachments

Slides from lecture; exercise sheet D.
Denotational semantics

\[ [-] : \text{Program source} \rightarrow \text{Mathematical structure.} \]

In denotational semantics we seek mathematical models for computation that free us from concrete details like syntax and scope rules; the hope is to reach the ‘idea’ behind a program. As ever compositionality is vital:

\[ [\text{if } B \text{ then } P \text{ else } Q] = \text{cond}([B], [P], [Q]). \]

This means that \([P]\) must include information about any contribution \(P\) may make to an enclosing program.

What is the right mathematics?

Set theory is the obvious choice, and seems able to handle integers, booleans, pairs, lists, functions, \ldots

But how should we treat the following?

- **Loops:** \(\text{for } (i = 0; i < 10; i++) \{ \text{code} \}\)
- **Recursive functions:**
  \[
  \text{fun } \text{hcf} \ x \ y = \text{if } x = 0 \text{ then } y \text{ else } \text{hcf}(y \text{ mod } x) \ x
  \]
- **Recursive datatypes:**
  \[
  \text{datatype } \text{tree} = \text{branch of } \text{tree} \times \text{tree} \\
  | \text{leaf of } \text{tree} \rightarrow \text{tree}
  \]

The problem lies in the interaction between finite programs and their potentially infinite behaviour.

Domain theory

‘Domains’ are mathematical objects with enough structure to support

- recursively defined values;
- recursively defined domains.

This is achieved through notions of

- finite elements, ordered by partial information;
- infinite elements, as the limit of finite ones;
- continuous maps, to preserve this structure.
References

Davey & Priestley give a gentle introduction to posets and domains. The books of Winskel, Gunter and Mitchell all cover denotational semantics and domain theory in some depth. Pitts’s course notes are recommended.

At a more advanced level, Plotkin’s ‘Pisa’ notes and the Handbook chapter of Abramsky & Jung cover almost everything there is to know about domains. Both are available electronically from Hypatia.

Posets

A partially ordered set or poset comprises a set $P$ and a binary relation `$\leq$' that is reflexive, transitive and antisymmetric:

$$x \leq y \land y \leq x \implies x = y.$$  

Small finite posets can be represented by their Hasse diagram.

For us `$\leq$' represents increasing information content.

Least upper bounds

A poset is pointed if it has a least element. When this exists it is unique and called bottom, written `$\bot$'.

An upper bound for a subset $A \subseteq P$ is an element $x \in P$ that is above every $a \in A$.

A least upper bound for $A$ may exist. If so then it is unique, written $\bigsqcup A$ and sometimes called the lub, supremum or join of $A$.
Cpo’s

An \(\omega\)-chain in a poset is an ascending sequence of elements

\[ x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots \]

A poset is \(\omega\)-chain complete if every \(\omega\)-chain has a least upper bound \(\bigsqcup_{n<\omega} x_n\). Such a poset is called an \(\omega\)-cpo, a cpo, or a domain.

The word domain is almost always "locally bound".

Maps preserving structure

A map \(f : P \rightarrow Q\) of posets is monotone if \(x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)\).

A map \(f : D \rightarrow E\) of cpo’s is continuous if it is monotone and preserves lubs:

\[ f \left( \bigsqcup d_n \right) = \bigsqcup f(d_n) . \]

A map between pointed cpo’s (‘cppo’s’) is strict if it preserves bottom \(f(\bot_D) = \bot_E\). We write \(f : D \rightarrow E\) for this.

Fixed point theorem

Any continuous function \(f : D \rightarrow D\) in a pointed cpo possesses a least fixed point, i.e. an element \(\text{fix}(f) \in D\) such that

- \(f(\text{fix}(f)) = \text{fix}(f)\);
- \(\forall d \in D . f(d) = d \Rightarrow \text{fix}(f) \sqsubseteq d\).

The proof is constructive and gives:

\[ \text{fix}(f) = \bigsqcup_{n<\omega} f^n(\bot) . \]
Scott induction

A subset $S \subseteq D$ of a pointed cpo is admissible if it contains $\bot$ and is closed under taking lubs of $\omega$-chains:

$$d_0 \subseteq d_1 \subseteq \cdots \text{ in } S \implies \bigcup_{n<\omega} d_n \in S \text{ too.}$$

Suppose $f : D \to D$ is continuous and $S \subseteq D$ admissible. To prove that $\text{fix}(f) \in S$ it is enough to show that $fS \subseteq S$:

$$(\forall d \in D . \ d \in S \Rightarrow f(d) \in S) \implies \text{fix}(f) \subseteq S.$$  

This is fixed point induction or Scott induction.

---

Domain constructors: Lifting

We lift a cpo $D$ by adding a least element, giving cppo $D_\bot$.

$$f : D \to E \quad \quad \quad up : D \to D_\bot$$

$$f_\bot : D_\bot \to E_\bot \quad \quad \quad d \mapsto [d]$$

Any set $X$ can be regarded as a discrete cpo, and lifting $X_\bot$ gives the corresponding flat cpo.

Conversely, any pointed cpo $D$ may be lowered by removing its least element to give $D_\bot$.

---

Domain constructors: Sums

The disjoint sum $D + E$ of two cpo's is simply their disjoint union, with $\subseteq_{D+\bot} = \subseteq_D \cup \subseteq_E$.

On maps:

$$f : D \to D' \quad g : E \to E'$$

$$f + g : D + E \to D' + E'$$

$$\text{intl} : D \to D + E$$

$$p : D \to F \quad q : E \to F$$

$$[p, q] : D + E \to F$$

$$\text{intr} : E \to D + E$$

The coalesced sum $D \oplus E$ of two cpo's identifies their bottom elements. For its action on maps, replace ‘$-$’ above with the strict ‘$\rightarrow$’.
Domain constructors: Products

The cartesian product of cpo's:

\[ D \times E = \{ (d, e) \mid d \in D, e \in E \} \]

\[ (d, e) \subseteq_{D \times E} (d', e') \quad \text{def} \quad d \subseteq_D d' \quad \& \quad e \subseteq_E e' \]

\[ \bigsqcup_{D \times E} (d_n, e_n) = \left( \bigsqcup_D d_n, \bigsqcup_E e_n \right) \]

\[ f : D \to D' \quad g : E \to E' \]

\[ f \times g : D \times E \to D' \times E' \]

\[ f : D \times E \to D \]

\[ g : D \to D' \quad p : F \to D \quad q : F \to E \]

\[ (p, q) : F \to D \times E \]

\[ \text{fst} : D \times E \to D \quad \text{snd} : D \times E \to E \]

On pointed cpo's the smash product is \( D \oplus E \cong (D_1 \times E_1)_1 \), with a corresponding action on strict maps.

6-13

Domain constructors: Function space

The set of continuous functions between any two cpo's \( D \) and \( E \) is itself a cpo, written \( E^D \) or \( (D \to E) \).

\[ f \subseteq g : E^D \quad \text{def} \quad \forall d \in D : f(d) \subseteq_E g(d) \]

\[ \left( \bigsqcup_{E^D} f_n \right)(d) = \bigsqcup_E f_n(d) \quad d \in D \]

\[ \text{apply} : E^D \times D \to E \]

\[ f : D \times E \to F \]

\[ \text{curry}(f) : D \to F^E \]

For pointed \( D \) and \( E \) there is also a cppo of strict functions \( (D \circ \to E) \).

6-14
References

The limit/colimit method for solving recursive domain equations originated with Scott. The following article is the classic reference for this technique.


Other work on domains mentioned in the lecture can be found in these sources.


Attachments

Slides from lecture; copy of the article by Smyth and Plotkin.
Domain Equations

&

The Interpretation of PCF

Recursively defined domains

Domain equations are a concise way to specify the properties desired of some domain.

\[ D \equiv D \perp \quad \text{vertical natural numbers} \]
\[ D \equiv \Sigma \oplus D \perp \quad \text{lazy natural numbers} \]
\[ D \equiv (A \times D) \perp \quad \text{\(\lambda\)-strings under prefix order} \]
\[ D \equiv D \rightarrow D \perp \quad \text{untyped \(\lambda\)-calculus} \]
\[ R \equiv S \rightarrow (S \oplus (S \otimes R)) \perp \quad \text{resumptions for interleaving of parallel processes} \]

Embeddings and projections

What we need is an analogue of the fixed-point theorem, but for domains themselves.

An embedding-projection pair \((e, p) : D \rightarrow E\) between two cpo’s is a pair of continuous maps

\[
\begin{array}{c}
D \xrightarrow{e \circ p} E \\
\quad \text{with } e \circ p \subseteq \text{id}_E \text{ and } p \circ e = \text{id}_D.
\end{array}
\]

These \(e,p\) pairs form a category \(\text{Cpo}^e\); for pointed domains we have \(\text{Cppo}^e\) which is the same as \(\text{Cppo}_L^e\).

An \(\omega\)-chain of embeddings is called an expanding sequence.

\[ D_0 \rightarrow D_1 \rightarrow D_2 \rightarrow \cdots \]
Limit/colimit coincidence

For any expanding sequence \( D_0 \rightarrow D_1 \rightarrow \cdots \) of cpo’s there is a unique cpo \( D \) that is both a limit and a colimit.

This bilimit also has the property that
\[
\bigsqcup_{n \in \omega} (D \xrightarrow{p_n} D_n \xrightarrow{e_n} D) = (D \xrightarrow{id} D).
\]

Covariant domain equations

All of the operators \((-)_\perp, \times, +, \otimes\) and \(\oplus\) act continuously on embeddings too. Given then a domain equation like
\[
D \equiv (\Sigma \oplus D_\perp) \overset{def}{=} \Phi(D)
\]
we have a functor \(\Phi : \text{Copro}_\perp \rightarrow \text{Copro}_\perp\) and can construct an expanding sequence of approximate solutions:
\[
1 \rightarrow \Phi(1) \rightarrow \Phi^2(1) \rightarrow \cdots.
\]
This has a bilimit \(\text{FIX}(\Phi)\) which solves the equation
\[
\text{FIX}(\Phi) \equiv \Phi(\text{FIX}(\Phi)).
\]

General domain equations

Function spaces \(\rightarrow, \circ : \text{Copro}_\perp \times \text{Copro}_\perp \rightarrow \text{Copro}_\perp\) are contravariant in their first component. However the dual nature of embeddings and projections means that for any such mixed variance functor
\[
\Phi : \text{Copro}_\perp \times \text{Copro}_\perp \rightarrow \text{Copro}_\perp
\]
there is an induced covariant functor
\[
\bar{\Phi} : \text{Copro}_\perp \times \text{Copro}_\perp \rightarrow \text{Copro}_\perp
\]
\[
((e, p), (e', p')) \mapsto (\Phi(p, e'), \Phi(e, p')).
\]
We can then take \(\text{FIX}(\Phi)\) to be the bilimit of
\[
1 \rightarrow \bar{\Phi}(1, 1) \rightarrow \bar{\Phi}(\bar{\Phi}(1, 1), \bar{\Phi}(1, 1)) \rightarrow \cdots.
\]
Methods for solving domain equations

- **Limit/colimit** Freyd’s work on *algebraically compact* categories generalises what happens here. From this Pitts has devised mixed induction/coinduction techniques for proving properties of recursively defined domains.

**Information systems** An alternative presentation of domains that builds them from consistent sets of tokens (see Winskel §12).

**Universal domains** It is possible to identify a domain into which all others embed, and then solve domain equations there (see Gunter §8.10.2).

\(^1\)For a suitable definition of ‘all’

---

**Domain model for PCF**

Types of PCF are interpreted by ccpo’s:

\[
D_\bot = \mathbb{B} \quad D_1 = \mathbb{N} \quad D_{\sigma \rightarrow \sigma'} = D_\sigma \rightarrow D_{\sigma'}.
\]

A type environment \(\Gamma\) is a list of typed variables. This is interpreted by a product domain:

\[
D_\Gamma = D_{\sigma_1} \times D_{\sigma_2} \times \cdots \times D_{\sigma_k} \quad \text{where } \Gamma = [x_1^{\sigma_1}, \ldots, x_k^{\sigma_k}].
\]

A term \(M : \sigma\) with free variables in \(\Gamma\) is interpreted by a map

\[
[\Gamma \vdash M : \sigma] : D_\Gamma \rightarrow D_\sigma.
\]

On closed terms this becomes

\[
[M : \sigma] \in D_\sigma.
\]

---

**Domain model for PCF continued**

The interpretation of PCF terms is given inductively over their structure:

\[
[\Gamma \vdash x_i^{\sigma_i} : \sigma_i] = \pi_i : D_\Gamma \rightarrow D_{\sigma_i}
\]

\[
[\Gamma \vdash \lambda x^{\sigma} . M : \sigma \rightarrow \sigma'] = \text{curry}(\{[\Gamma \vdash M : \sigma']\})
\]

\[
[\Gamma \vdash MN : \sigma] = \text{apply} \circ \{[\Gamma \vdash M : \sigma'], [\Gamma \vdash N : \sigma]\}
\]

\[
[\Gamma \vdash \mu x^{\sigma} . M : \sigma] = \text{fix} \circ \{[\Gamma \vdash \lambda x^{\sigma} . M : \sigma \rightarrow \sigma]\}
\]

where \(\text{fix} : D^D \rightarrow D\) enacts the fixed point theorem. On constants:

\[
\text{[succ]}(x) = \begin{cases} 
[n + 1] & \text{if } x = [n] \\
\bot & \text{if } x = \bot 
\end{cases}
\]

\([\text{tt}] = [\text{true}] \in \mathbb{B} \quad \text{etc.}\)
Static properties of the model

**Well-defined** The function $[\Gamma \vdash M : \sigma] : \mathcal{D}_\Gamma \to \mathcal{D}_\sigma$ is continuous.

**Compositional** For any context $\mathcal{C}[-]$

\[
[\mathcal{C}[M_1]] = [\mathcal{C}[M_2]] \implies [\mathcal{C}[\Gamma M_1]] = [\mathcal{C}[\Gamma M_2]].
\]

**Substitutive** For term $M$ with variable $x^\sigma$ and term $N$ of type $\sigma$:

\[
[M[N/x^\sigma]] = [x^\sigma \vdash M [\{N\}]].
\]

All proofs are by appropriate forms of induction.

Variations: lazy or strict

We can capture the two alternative semantics for pairs in PCF:

- $\mathcal{D}_\sigma \times \sigma' = \mathcal{D}_\sigma \times \mathcal{D}_{\sigma'}$ lazy pairing; or
- $\mathcal{D}_\sigma \times \sigma' = \mathcal{D}_\sigma \otimes \mathcal{D}_{\sigma'}$ strict pairing.

For call-by-value VPCF we rework the model:

\[
\begin{align*}
V_0 &= \mathcal{B} & V_1 &= \mathcal{N} & V_{\sigma \times \sigma'} &= V_\sigma \to (V_{\sigma'})_\perp \\
V_\Gamma &= V_{\sigma_1} \times \cdots \times V_{\sigma_k} & [\Gamma \vdash M : \sigma]_\nu : V_\Gamma \to (V_{\sigma})_\perp.
\end{align*}
\]

Variations: state, storage and I/O

For VPCF with state we change again:

\[
V_{\sigma \to \sigma'} = (S \times V_\sigma) \to (S \times V_{\sigma'})_\perp
\]

\[
[\Gamma \vdash M : \sigma]_\nu : (S \times V_\Gamma) \to (S \times V_{\sigma})_\perp.
\]

Different forms of state demand different denotations:

<table>
<thead>
<tr>
<th>Storage</th>
<th>$S = \text{Loc} \to V_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input/output</td>
<td>$S = V_{i\sigma} \times V_{o\sigma}$</td>
</tr>
<tr>
<td></td>
<td>where $V_{i\sigma} \equiv (V_1 \times V_{o\sigma})_\perp$.</td>
</tr>
</tbody>
</table>

Using monads can give a uniform approach to extensions like these.
Domain model for IV

For an imperative language like IV we take
\[ D_{\text{int}} = \mathbb{Z} \quad D_{\text{bool}} = \mathbb{B} \quad D_{\text{comm}} = S \rightarrow S \perp \quad S = \text{IntVar} \rightarrow D_{\text{int}} \]
or if command variables are included:
\[ S = (\text{IntVar} \rightarrow D_{\text{int}}) \times (\text{CommVar} \rightarrow D_{\text{comm}}). \]
The denotation of terms is straightforward, giving
\[ [I] \in D_{\text{int}} \quad [B] \in D_{\text{bool}} \quad [C] \in D_{\text{comm}} \]
with a fixed point on \( D_{\text{comm}} \) used to interpret repeat statements.

Other domain models

Concurrency Abramsky used the equation
\[ D = \mathcal{P}(A \times D) \]
to give a model of CCS, with equality in the model matching bisimulation.

Dataflow The Kahn principle determines the behaviour of a dataflow network as a least fixed point in a suitable domain.

Integration Edalat has given a domain-based description of integration and so obtained efficient algorithms for fractal image decompression.
References

Plotkin (1977) describes the domain model for PCF, demonstrates the problem with parallel or, and shows that the model is fully abstract for PCF extended with a parallel conditional $\text{pif}$. The following show how to cut down the model using logical relations.


Some references on the ‘Further Directions’ mentioned in the lecture.


Attachments

Slides from lecture; copy of the article *Semantics of Interaction* by Abramsky.
Semantics of Computation — Lecture 8

Properties of the Domain Model for PCF
&
Further Directions in Semantics

Soundness

The domain interpretation of PCF is sound (or correct) with respect to the operational semantics:

\[ M \downarrow V \Rightarrow [M] = [V] \in D_\sigma \]
\[ M \rightarrow M' \Rightarrow [M] = [M'] \in D_\sigma \]

where \( M, M' \) and \( V \) are closed expressions of type \( \sigma \). Proof is by rule induction, and a corollary is that for any program \( P \):

\[ [P] = \bot \Rightarrow P \Uparrow. \]

Adequacy

The model is also adequate, meaning that for any program \( P \):

\[ [P] \neq \bot \Rightarrow P \Downarrow. \]

This is a converse to soundness and is significantly harder to prove. As a corollary though we have that at every type

\[ [M] = [M'] \Rightarrow M \equiv_{ctx} M' \]

which allows us to use the domain model for reasoning about contextual equivalence.
Proof of adequacy

The key element is a logical relation between elements \( d \in D_\sigma \) and closed expressions of type \( \sigma \). This is defined inductively over types:

\[
\begin{align*}
\text{d} \lessdot_B \text{B} & \iff d = \bot \text{ or } d = \text{true} \& B \downarrow \text{tt} \\
& \quad \text{or } d = \text{false} \& B \downarrow \text{ff} \\
\text{d} \lessdot_i \text{N} & \iff d = \bot \text{ or } d = [0] \& N \downarrow \text{0} \\
& \quad \text{or } \ldots \\
\text{d} \lessdot_{\sigma \rightarrow \tau} \text{F} & \iff \forall e \lessdot_\sigma \text{M} . \text{de} \lessdot_{\sigma'} \text{FM} 
\end{align*}
\]

It can be shown that

\[
\text{d} \lessdot_\sigma \text{M} \iff d \subseteq [\text{M}] \in D_\sigma.
\]

In particular \([\text{M}] \lessdot \text{M} and adequacy then follows from the explicit form of \( \lessdot_\sigma \) at ground types.

Full abstraction

Adequacy allows us to prove contextual equivalence with the domain model, but how good is this?

The desirable property

\[
\text{M} \equiv_{\text{ctx}} \text{M'} \implies [\text{M}] = [\text{M'}]
\]

is called full abstraction.

One route to proving full abstraction would be a definability result saying that every element \( d \in D_\sigma \) is equal to \([\text{M}]\) for some term \( \text{M} : \sigma \).

For the basic domain model this is unfortunately not true . . .

Parallel or

Consider the element \( \text{por} \in D_{\sigma \rightarrow \tau \rightarrow \tau} \) defined by

\[
\text{por} \ b_1 \ b_2 \overset{\text{def}}{=} \begin{cases} 
\text{true} & \text{if } b_1 = [\text{true}] \text{ or } b_2 = [\text{true}] \\
\text{false} & \text{if } b_1 = b_2 = [\text{false}] \\
\bot & \text{otherwise}
\end{cases}
\]

and the PCF term

\[
\text{Test} \overset{\text{def}}{=} \lambda b. \text{if}_\sigma \{ (\text{if} \ \Omega) \land (\text{if} \ \text{tt} \ \text{tt}) \land \text{not}(\text{if} \ \text{ff} \ \text{ff}) \} \ b \ \Omega
\]

where \( \Omega = \mu x. x^{\sigma} \) diverges. Then \([\text{Test}] \overset{\text{def}}{=} b \text{ por} = b \) and so

\[
[\text{Test} \ \text{tt}] \neq [\text{Test} \ \text{ff}]
\]

but

\[
\text{Test} \ \text{tt} \equiv_{\text{ctx}} \text{Test} \ \text{ff}
\]

because there is no PCF term with the behaviour of \( \text{por} \) (proof by another logical relation).
**Sequentiality**

The central difficulty is that every program in PCF is clearly sequential in some sense, but the domains do not capture this. Solutions?

**Go back to syntax** The term model is known to be fully abstract; but this syntactic result gives little semantic insight.

**Reduce the model** Logical relations can discard undefinable elements, leaving a fully abstract model.

**Enhance the domains** Stable domain theory, concrete data structures, …

**Change the language** The model is fully abstract for PCF+\pi.

**Devise a new model** Game semantics is one.

---

**Polymorphism**

Often in a programming language it is convenient to have functions that can operate at more than one type.

**Ad-hoc polymorphism** For example + : int \rightarrow int \rightarrow int and + : float \rightarrow float \rightarrow float, where the underlying algorithms are unrelated. The type classes of Haskell address this.

**Parametric polymorphism** This is where the same algorithm can be used at several types, as with this identity function:

$$\lambda A : A. : A : A$$

A related issue is that of subtyping $\sigma \leq \tau$ where values of one type can be used at another. (Subtyping is not inheritance.)

---

**Type systems**

The polymorphic $\lambda$-calculus or System F can be used to encode several standard type constructors. For example:

$$A \times B \overset{\text{def}}{=} \forall C . (A \rightarrow B \rightarrow C) \rightarrow C$$

$$\text{fst} \overset{\text{def}}{=} \forall A. \forall B. \forall \alpha. A \times B . \alpha P A B . \alpha P A$$

$$A + B \overset{\text{def}}{=} \forall C . (A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C.$$

This is just one of the type systems in Barendregt's $\lambda$-cube. Others like the Calculus of Constructions allow abstraction over type constructors and dependent types.
Linear logic

Due to Girard, linear logic is a more fine-grained version of traditional logic, with a particular emphasis on the mechanism of proof.

The standard constants false, true, ∧, ∨, ¬, and ⇒ become

0, 1, ⊗, ⊕, ⊥, T, &,#, (¬)^, →, !, ?

For example the linear implication A ⇒ B means that B can be deduced using A exactly once. Traditional implication is then obtained by A ⇒ B = !A ⇒ B where the exponential !A provides arbitrarily many copies of A.

Game semantics

This approach considers games where a player (the program) and an opponent (the environment) alternately make moves from some specified set. There are several operations on games: in A ⊗ B both games A and B are played simultaneously; in A ⇝ B the player must play as opponent in A and as player in B.

Maps s : A ⇒ B between games are given by strategies for playing (A ⇒ B). There may be some constraints on strategies: that they be winning, history-free, innocent and so forth.

Developed by Hyland/Ong: Abramsky/Jagadeesan/Malacaria: Nickau

Models for interaction

Game semantics is part of a current movement towards a treatment of computation as interaction. The following are some active research programs in this area.

- Presheaf semantics (Winskel)
- Action calculi (Milner)
- Interaction categories (Abramsky)
- Geometry of interaction (Girard)

These all seek to integrate programs, logic, concurrency and communication within a common semantic framework.
Further resources

Here is a somewhat arbitrary selection of the many conferences and journals whose coverage includes semantics of computation.

- LICS (Logic in Comp. Science)
- Concur (Concurrency Theory)
- ICALP (Annual conf. of EATCS)
- Information & Computation
- Theoretical Computer Science
- Journal of Functional Programming

The multiple volumes of the Handbook of Theoretical Computer Science and the Handbook of Logic in Computer Science are both fine reference texts in this area.
Semantics of Computation

Exercise sheet A — Induction and PCF

1. Show that for any binary tree T the size(T) < 2^{\text{depth}(T)}. A tree is balanced if every branch has the same depth; equivalently, if both subtrees are balanced and of the same depth. Show that for any balanced tree B the size(B) = 2^{\text{depth}(B)} - 1.

[Use structural induction to prove the proposition \( \phi(T) \overset{\text{def}}{=} (T \text{ balanced } \Rightarrow \text{size}(T) = 2^{\text{depth}(B)} - 1).]\]

2. Consider the alphabet \( \Sigma = \{a, b\} \) and the set \( \Sigma^* \) of finite strings over it. Give inductive definitions for the following subsets.

(a) \( P \subseteq \Sigma^* \), the palindromes.
(b) \( E \subseteq \Sigma^* \), the strings of even length.
(c) \( R \subseteq \Sigma^* \times \Sigma^* \), pairs of strings with their reversals, i.e. the set \( \{ (s, t) \mid s = \text{reverse}(t) \} \).

3. Use if\(_o\) to write PCF functions for not, and and or operating on booleans:

\[
\text{not} : o \rightarrow o \quad \text{and} : o \rightarrow o \rightarrow o \quad \text{or} : o \rightarrow o \rightarrow o.
\]

Show that \( (\text{not} \, ff) \) has type \( o \). Write a PCF function \( \text{isone} : t \rightarrow o \) that tests for the number 1.

4. What are the free variables of the terms

\[
succ \, z, \ \lambda x.(\lambda y.x), \ x(\lambda y.\text{pred} \, y) \quad \text{and} \quad (\lambda x.y)x?
\]

What is \( \text{fv}(M[N/x]) \) in terms of \( \text{fv}(M) \) and \( \text{fv}(N) \)? Suggest an appropriate induction method that would prove your assertion.

5. Use rule induction to show the following facts about typing in PCF.

(a) If \( M : \sigma \) and \( M : \sigma' \) then \( \sigma = \sigma' \).
(b) If \( M : \tau \) and \( N : \sigma \) then \( M[N/x] : \tau \).

6. (Bonus box) Use mathematical induction to prove the converse to the ‘divisibility means divisibility’ proposition on slide 2-8.
Semantics of Computation

Exercise sheet B — Operational semantics I

1. The imperative language IV has the following types and term classes.

   Types \( t ::= \text{comm} | \text{int\hspace{1pt}exp} | \text{bool\hspace{1pt}exp} \)

   Commands \( C ::= \text{skip} | C_1; C_2 | x ::= I | C_1 \text{\textbf{if}} B \text{\textbf{else}} C_2 | \text{\textbf{repeat}} C \text{\textbf{until}} B \)

   Integer expressions \( I ::= x | n | I_1 \text{\textbf{iop}} I_2 \)

   Boolean expressions \( B ::= \text{true} | \text{false} | \text{not} B | B_1 \text{\textbf{bop}} B_2 | I_1 \text{\textbf{comp}} I_2 \)

   \hspace{1cm} \text{iop} \in \{+, -, \times\} \hspace{1cm} n \in \mathbb{Z}

   \hspace{1cm} \text{bop} \in \{\text{and, or}\} \hspace{1cm} x \in \text{Var}, \text{ a countably infinite set of integer variables}

   \hspace{1cm} \text{comp} \in \{<, >, =, \neq, \leq, \geq\}

Construct suitable typing rules for the language. Write some code to place the exponential \( x^y \) into variable \( z \) and show that it has type \text{comm} (you may assume that \( y \geq 1 \) and you need not preserve the values of \( x \) and \( y \)).

[ You would be correct in objecting that the language has as yet no formally defined semantics. However, this is traditionally no bar to writing programs. ]

2. Expand the two reduction sequences at the bottom of slide 4-1, and also the two at the bottom of slide 4-2.

3. Using pattern matching as in ML one might write the iteration function

   \[
   \text{fun iterate\hspace{1pt}f\hspace{1pt}0\hspace{1pt}x = x}
   \]

   \[
   \hspace{1cm} \mid \hspace{1cm} \text{iterate\hspace{1pt}f\hspace{1pt}n\hspace{1pt}x = \hspace{1pt} iterate\hspace{1pt}f\hspace{1pt}(\hspace{1pt}\text{pred\hspace{1pt}n\hspace{1pt}})\hspace{1pt}(\hspace{1pt}f\hspace{1pt}x\hspace{1pt})\hspace{1pt}}.
   \]

   Translate this into PCF with \( \mu \)-abstraction for recursion. What do you expect that the function \( \text{iterate succ : } t \rightarrow t \rightarrow t \) will do?

[ Bonus box: how will the length of reduction sequence for \( \text{iterate succ n m} \) depend on \( n \) and \( m \)? ]

4. Prove the \textit{subject reduction} property for PCF, \textit{i.e.} that reduction of terms preserves types.

5. Show that the following congruence rule is valid for Kleene equivalence.

   \[
   B_1 \equiv^{kl} B_2 \quad M_1 \equiv^{kl} M_2 \quad M'_1 \equiv^{kl} M'_2
   \]

   \[\text{if } B_1 M_1 M'_1 \equiv^{kl} \text{ if } B_2 M_2 M'_2\]

   Why are the terms \( F(\text{if } B \text{\hspace{1pt}M\hspace{1pt}M'}) \) and \( F(\text{if } \text{\hspace{1pt}BM\hspace{1pt}M'}) \) not Kleene equivalent in general?
Semantics of Computation

Exercise sheet C — Operational semantics II

1. Arriving back at your desk after Christmas, you find that someone had left a window open over the holiday and everything is buried in snow. The most severe damage is to your prized copy of the *IV Reference Manual*. You can’t afford to buy a new one\(^1\) and so must reconstruct the operational semantics of IV from the fragments remaining. Luckily the opening section on definitions is unharmed, but can you make good the damage to the remainder?

2. A pre-release semantics for IV included the following reduction rule:

\[
\begin{align*}
C & \rightarrow C' \\
\text{repeat } C \text{ until } B & \rightarrow \text{repeat } C' \text{ until } B
\end{align*}
\]

What is wrong with this?

3. “Integer and boolean expressions in IV always converge and never change the store”.

   (a) There are four assertions made here: write them out formally and indicate suitable proof methods for each.

   (b) Complete the proof that integer expressions always converge.

4. For version 1.3 of IV, its publishers *IntVar Corp.* are proposing to change the semantics of the boolean operations ‘or’ and ‘and’ to short-circuit versions that only evaluate their second argument if absolutely necessary. Give suitable new reduction and evaluation rules. Will this change be backward-compatible?

5. Define a notion of Kleene equivalence between IV configurations. Give three examples.

6. Suggest suitable notions of ‘program’ and ‘behaviour’ for the IV language, and derive a definition of contextual equivalence between terms

---

\(^1\)(this is just after Christmas, remember)
Definitions

An IV store is a function $s : \text{Var} \to \mathbb{Z}$ that is zero everywhere except for some finite set of variables. A configuration $\langle s, E \rangle$ is a store-term pair. The operational semantics is defined on configurations.

State convention: When store is simply passed from left to right in a rule, they need not appear explicitly in the configurations. Thus

$$E_1 \to E'_1 \quad E_2 \to E'_2 \quad \ldots \quad E_k \to E'_k \quad \overline{E \to E'}$$

means

$$\langle s, E_1 \rangle \to \langle s_2, E'_1 \rangle \quad \langle s_2, E_2 \rangle \to \langle s_3, E'_2 \rangle \quad \ldots \quad \langle s_k, E_k \rangle \to \langle s', E'_k \rangle \quad \overline{\langle s, E \rangle \to \langle s', E' \rangle}.$$ 

Transition Semantics

\[
\begin{align*}
C_1 & \to C'_1 \\
C_1 ; C_2 & \to C'_1 ; C_2 \\
I & \to I' \\
\overline{\text{skip} ; C \to I'} \\
B & \to B' \\
\overline{C_1 \text{ if } B \text{ else } C_2 \to C_1 \text{ if } B \text{ else } C_2 \to C_1} \\
\overline{\text{repeat } C \text{ until } B \to C \text{ ; (skip if } B \text{ else } \\
\overline{\text{false else}} \text{ )}} \\
\langle s, x \rangle & \to \langle s, n \rangle \quad (n = s(x)) \\
I_1 \text{ iop } I_2 & \to I'_1 \text{ iop } I'_2 \\
\overline{\text{not } b \to \text{ true, false}} \\
B_1 & \to B'_1 \\
\overline{b \text{ bop } B_2 \to b \text{ bop } B'_2 \quad b = \text{ true, false}} \\
I_2 & \to I'_2 \\
\overline{\text{not } B \to \text{ not } B} \\
I_2 & \to n \text{ comp } I'_2 \\
\overline{n_1 \text{ comp } n_2 \to \text{ true (if } n_1 \text{ comp } n_2)} \\
\overline{n_1 \text{ comp } n_2 \to \text{ false}} \\
\end{align*}
\]
Reduction Contexts

\[ R[\_] ::= [\_] | R[\_]; C \mid x := R[\_] \mid C_1 \text{ if } R[\_] \text{ else } C_2 \\
| R[\_]\ \text{iop}\ I \mid \text{n}\text{op}\ R[\_] \\
| \text{bop} \mid \text{true} \mid \text{comp} \mid \text{comp} \text{bop} \mid \text{false}\text{bop} \]

The redexes are the axioms of the reduction system, whose rules can then be summarised as follows:

\[ E \rightarrow E' \]

Evaluation Semantics

The IV values are \[\text{skip}, \text{n}, \text{true}, \text{false}\]: they all evaluate to themselves.

\[ C_1 \downarrow \text{skip} \quad C_2 \downarrow \text{skip} \]  
\[ C_1; C_2 \downarrow \text{skip} \]

\[ B \downarrow \text{true} \quad C_1 \downarrow \text{skip} \]
\[ C_1 \text{ if } B \text{ else } C_2 \downarrow \text{skip} \]

\[ C \downarrow \text{skip} \quad \text{repeat C until } B \downarrow \text{skip} \]

\[ \langle s, I \rangle \downarrow \langle s', n \rangle \]

\[ \langle s, x := I \rangle \downarrow \]

\[ \langle s, x \rangle \downarrow \]

\[ I_1 \text{ iop} I_2 \downarrow m \]

\[ b = b_1 \text{ bop} b_2 \]
\[ \text{not} B \downarrow b' \]

\[ I_1 \downarrow n_1 \quad I_2 \downarrow n_2 \]
\[ I \text{ comp} I_2 \downarrow b \]  
\[ b = \text{true if } n_1 \text{ comp} n_2 \text{ and other wise} \]
Semantics of Computation

Exercise sheet D — Operational Semantics III and Domain Theory

Thanks to your first-class knowledge of semantics, you have secured a job in the Research Division of IntVar Corp. Your team is devising an operational semantics for IV Version 2, which the Marketing Division has decided will be called “Visual IV++ Internet Edition”. They have also declared that it is to be object-oriented, multi-threaded, web-aware and with a sophisticated collaborative GUI. All that remains is to design the actual language. Your task is as follows.

1. Devise an operational semantics (evaluation or reduction) for a try/abort construction. Informally, the desired behaviour is that try C should run the command C as normal, but if this reaches an abort then the command is abandoned and control returns to the nearest enclosing try.

Demonstrate the termination of these two phrases in your semantics:

\[ \text{try (repeat abort until false)} \]
\[ \text{try (repeat skip until true)} \]

2. To make commands “first class” the new IV will supplement the existing integer variables \(x, y, \cdots \in \text{IntVar}\) with command variables \(a, b, c \in \text{CommVar}\). Construct a suitable operational semantics to support this.

What happens to this program in your semantics:

\[ a := (x := x + 1; a); a \]

Can you encode a replacement for repeat/until using command variables?

3. Draw Hasse diagrams for the following cpo’s:

\[ \mathcal{P}((a, b, c)) \quad \mathbb{B} \oplus \mathbb{B} \quad \mathbb{B} \times \mathbb{B} \quad \Sigma \rightarrow \Sigma. \]

Here \(\mathcal{P}(\mathcal{P})\) is the powerset, ordered by subset inclusion ‘\(\subseteq\)’.

4. An element \(x\) of a poset \(P\) is the least if \(x \subseteq y\) for every \(y \subseteq P\). Show that if \(P\) has a least element then it is unique.

5. Show that if \(f : D \rightarrow E\) and \(g : E \rightarrow F\) are continuous functions, then so is their composition

\[ g \circ f : D \rightarrow F \]
\[ d \mapsto g(f(d)). \]

(Recall that to show a function continuous, it is first necessary to demonstrate that it is monotone.)

6. Prove that the application map is monotone:

\[ \text{apply} : E^D \times D \rightarrow E \]
\[ (f, d) \mapsto f(d). \]

[ Bonus box: show that the curry operation is monotone, even continuous. ]